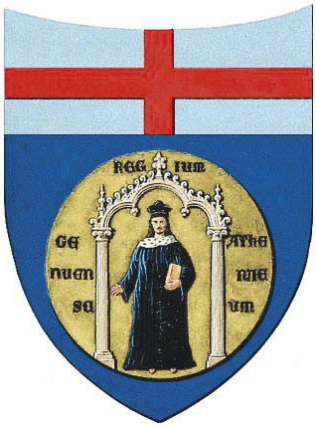
Università degli Studi di Genova

Facoltà di Ingegneria



Tesi di Laurea in Ingegneria Elettronica

Porting e analisi di sistemi RTOS: verifica sperimentale dei parametri caratteristici su scheda embedded con microcontrollore a basso costo.

Relatore:	Chiar.mo Prof. Ing. RODOLFO ZUNINO
Relatori aziendali:	Ing. ALDO SEBASTIANI
	Ing. MATTEO CANTARINI
Candidati:	DAVIDE LUCIANI
	MICHELE SEMINO

26 Settembre 2008

Anno Accademico 2007 – 2008

Abstract

"Porting and analysis of RTOS systems: experimental verification of characteristic parameters on an embedded platform with a low cost microcontroller"

In the last years the use of embedded electronic systems have grown in an exponential way and the great versatility has made embedded systems useful for several applications.
Among those devices, microcontrollers (µC) have become more and more complex and sophisticated; this trend of production has made necessary the use of custom software called RTOS to manage the microchip in order to provide an abstraction of it.
The goal of our thesis is to analyze some parameters of those systems to find, at the end, which of those should be the favourite RTOS compatibly with our applications.

Alla Commissione di Laurea e di Diploma

Alla Commissione Tirocini e Tesi

Sottopongo la tesi redatta dallo studente Davide Luciani dal titolo ”Porting e analisi di sistemi RTOS: verifica sperimentale dei parametri caratteristici su scheda embedded con microcontrollore a basso costo”.
Ho esaminato, nella forma e nel contenuto, la versione finale di questo elaborato scritto, e propongo che la tesi sia valutata positivamente assegnando i corrispondenti crediti formativi.

Il Relatore Accademico

Prof. Ing. Rodolfo Zunino

Ringraziamenti

Innanzitutto apriamo questa pagina ringraziando il Chiar.mo Prof. Ing. Rodolfo Zunino la cui disponibilità e il cui interesse nei nostri riguardi sono sempre stati vivi e cordiali, facendo nascere in noi una vera passione per la programmazione embedded.
Il nostro grande grazie va anche soprattutto all'Ing. Matteo Cantarini e all'Ing. Luigi Veardo di ELSAG Datamat i quali ci hanno accompagnati con immensa disponibilità passo passo in questo affascinante mondo mostrando una partecipazione costante e fornendoci una soluzione in ogni momento di difficoltà.

Davide e Michele.

Il primo grazie va senza dubbio a Mio Padre che, sin dall'inizio, ha creduto in tutto questo impiegando tutti i mezzi a sua disposizione, e forse anche qualcosa di più, per la buona riuscita dell'opera. Spero proprio di non averlo deluso.
Il secondo grazie va a Mimmo Barlocco che in tutti questi anni, oltre ad un grande maestro di vita, è stato per me come un secondo padre, consigliandomi sempre nel bene e dandomi la possibilità di poter seguire la mia strada con molta più tranquillità.
Grazie a Nadia Bassignani, che ogni volta in cui ha potuto, ha fatto il possibile per farmi sentire a casa, sino dai tempi del liceo; come ti dico sempre: "Nadia sei unica".

Dopodichè il mio grazie, ma soprattutto il mio abbraccio forte, va a Vito, Micky, Frank, Takke, Skeno, Ilir ed Edo, componenti inseparabili del Gruppo di Studi, con i quali ho condiviso la maggior parte del mio tempo negli ultimi anni; tra questi stringo forte Micky, amico e compare di avventure (tra le ultime questa tesi), che rimprovero spesso per la sua "eccessiva estroversione" ma quando ci ripenso, sorrido. La strada è stata ardua e piena di ostacoli, ma abbiamo sempre tenuto duro e, pian piano, siamo arrivati a questa importante destinazione che, sì intermedia, ma ci siamo arrivati. Grazie di cuore Amici.

Che altro dire. La strada è ancora lunga. Forza ragazzi.

Davide.

Prefazione

Negli ultimi anni è andato via via sviluppandosi sempre più l'utilizzo di apparati elettronici embedded relativamente alle più svariate applicazioni.

Tra questi, un'importanza particolarmente rilevante va associata alla vertiginosa diffusione dei microcontrollori (μC) i quali, grazie alla loro versatilità applicativa ed il costo particolarmente ridotto, hanno dato origine ad una vera e propria esplosione commerciale anche in ambiti dove essi non erano strettamente necessari.

Con lo sviluppo esponenziale delle tecnologie produttive, questi oggetti sono diventati sempre più sofisticati e conseguentemente sempre più complessi; da qui nasce la necessità di gestire le loro funzionalità attraverso astrazioni software fornite dai Real-Time Operating Systems (RTOS) che, dovendo però abbracciare i più svariati ambiti applicativi, vengono sviluppati in modo "customizzato" ovvero ottimizzati per specifiche applicazioni pur mantenendo un'interfaccia di tipo generale relativamente al μC.

A seconda delle finalità dei gruppi di sviluppo di RTOS, questi possono quindi avere caratteristiche più o meno ottimizzate rendendoli di fatto assai diversi tra loro; infatti navigando in rete è possibile notare come ciascuno di questi sia particolarmente incentrato su funzionalità specifiche che spaziano dalla rapidità di azione dello scheduler alla completa gestione delle attività di networking.

Lo scopo della nostra tesi è dunque quello di individuare quali parametri dell'eventuale RTOS scelto siano direttamente coinvolti al nostro progetto; ci occuperemo quindi di analizzare e valutare alcuni parametri fondamentali relativi a RTOS candidati a supportare la nostra applicazione.

Siamo partiti infatti da una cerchia di sette RTOS reperiti in rete aventi caratteristiche, a prima vista, compatibili con le nostre richieste prestazionali; dopodichè è seguita una più accurata fase di lettura e analisi dei relativi datasheets che ci ha portati a restringere il cerchio di interesse soltanto a due RTOS in quanto mostravano, relativamente ad alcuni parametri, performance indicate per la nostra applicazione: FreeRTOS e IAR PowerPac.

A queste prime due fasi, ne è seguita una terza che ha previsto alcune sessioni di test atte a valutare sul campo le prestazioni del RTOS relativamente a tre parametri specifici che avrebbero successivamente individuato, tra i due, il più adatto ai nostri scopi applicativi.

Per poter effettuare i tests essenziali sui RTOS necessitiamo quindi di un hardware in grado di accoglierli a bordo. Scelto quindi il microcontrollore più adatto al nostro scopo ovvero l'NXP LPC2378 presso i laboratori di ELSAG Datamat, abbiamo individuato ed analizzato, grazie al supporto degli ingegneri Matteo Cantarini e Luigi Veardo, le performance operative dei due RTOS sopraccitati in maniera tale da poter avere elementi sufficienti ad individuare quale potesse essere il sistema a noi più vicino.

I parametri che abbiamo stabilito essere direttamente coinvolti nella scelta tra FreeRTOS e IAR PowerPac sono stati: Context Switch Time (CST), Interrupt Latency e Footprint.

La nostra applicazione necessita di una particolare velocità operativa e di un'esigua occupazione di memoria; da qui l'assoluta necessità di avere sia un tempo di commutazione di contesto sia una latenza nella gestione delle interruzioni adeguatamente ridotti nonché di avere in memoria un'immagine del sistema che sia la più ridotta possibile.

Ciascuno di questi parametri è stato quindi calcolato ed approfondito singolarmente per ciascuno dei due RTOS messi a confronto.

Tra i due, uno è risultato avere performance superiori rispetto al diretto concorrente ed è quindi stato scelto da noi come "il sistema".

INDICE

1. Introduzione

1.1. I Sistemi Embedded

Qualcuno si sarà sicuramente chiesto come diavolo funzioni il proprio telefono cellulare o la propria lavatrice oppure il sistema frenante ABS della propria auto. Ebbene tutto ciò si può riassumere con due sole parole: Sistemi Embedded.
Letteralmente il termine Embedded in Italiano significa *incastonato* ed effettivamente è proprio l'appellativo che meglio può descrivere il modo di impiego di tali sistemi.
Infatti, in elettronica, con il termine Sistema Embedded si identificano genericamente dei sistemi elettronici a Microprocessore (μP) progettati appositamente per una determinata applicazione, spesso con un determinato hardware ad hoc, integrati nel sistema che controllano e in grado di gestirne tutte o parte delle funzionalità.
Contrariamente ai sistemi General Purpose, categoria a cui appartengono ad esempio i nostri computer di casa, i Sistemi Embedded devono svolgere dei compiti conosciuti già durante la fase della loro progettazione; ad esempio quando viene prodotto un Pentium 4, la Intel (casa produttrice di tale μP) non si pone il problema del fatto che esso venga usato per scopi di videoscrittura piuttosto che per ascoltare un Mp3; al contrario i progettisti della Motorola sapranno a priori l'applicazione che il loro μP dovrà coprire e di conseguenza lo progetteranno in maniera tale da funzionare al meglio per la suddetta applicazione [1].
Grazie a ciò l'hardware può essere ridotto ai minimi termini per ottimizzarne lo spazio occupato, i consumi ed il costo di fabbricazione.
Inoltre su questi dispositivi viene eseguito del software che risiede sulla memoria stessa dell'apparato e, nella maggior parte dei casi, l'esecuzione è in tempo reale (Real-Time); questo permette un controllo deterministico dei tempi di esecuzione.

1.2. Breve storia di tali sistemi

È curioso vedere come la presenza di Sistemi Embedded, che è andata sviluppandosi sempre più negli ultimi anni, fosse già esistente, anche se in forma più grezza, all'inizio degli anni sessanta specificatamente in ambito aerospaziale.

Nel 1961, infatti, compare quello che si può definire il primo vero Sistema Embedded prodotto in massa; esso si occupava della guida dei missili Minuteman tecnicamente conosciuto come Autonetics D-17, che utilizzava circuiti logici con transistor ed un hard disk come memoria principale.
In seguito, nei successivi decenni, i Sistemi Embedded hanno subito una riduzione dei costi così come un'enorme crescita della loro capacità di calcolo e delle loro funzionalità.
Il primo μP progettato per essere messo in commercio è stato l'Intel 4004, che venne montato su calcolatrici ed altri sistemi di piccole dimensioni. Esso richiedeva, comunque, dei chip di memoria esterni ed altra logica di supporto.
Verso la fine degli anni settanta, i microprocessori ad 8 bit erano la norma, ma necessitavano sempre di memoria esterna, logica di decodifica e di un interfaccia con il mondo esterno. In ogni caso, i prezzi erano in caduta libera e sempre più applicazioni cominciarono così ad adottare questo approccio, piuttosto che metodologie di progetto di circuiti logici personalizzate.
Verso la metà degli anni ottanta, un maggiore grado di integrazione permise il montaggio di altri componenti, in precedenza collegate esternamente, sullo stesso chip del processore. Questi sistemi integrati vennero chiamati microcontrollori (μC) piuttosto che microprocessori e fu possibile la loro utilizzazione di massa. Con un così basso costo per componente, questa alternativa divenne molto più interessante che costruire interamente circuiti logici dedicati. Ci fu un'esplosione del numero di sistemi embedded distribuiti sul mercato, così come delle componenti fornite da vari produttori per facilitare la progettazione di tali sistemi.
Parallelamente ai μC vennero alla luce i DSP (Digital Signal Processors) e i PLD (Programmable Logic Devices); mentre i primi erano ottimizzati per implementare in maniera estremamente efficiente algoritmi relativi al condizionamento di segnali digitali, i secondi erano dispositivi elettronici con logica configurabile previa programmazione.
Verso la fine degli anni ottanta, i sistemi embedded rappresentavano la regola piuttosto che l'eccezione per quasi tutti i dispositivi elettronici, tendenza che continua tuttora.

1.2.1. Digital Signal Processor (DSP)

Come accennato al paragrafo precedente, quando si tratta di elaborazione di segnali digitali bisogna dotarsi del dispositivo hardware adeguato ovvero il DSP.

Questo microprocessore è ottimizzato per eseguire in maniera estremamente efficiente sequenze di istruzioni ricorrenti, come ad esempio somme, moltiplicazioni e traslazioni nel condizionamento dei segnali.

A questo proposito, i DSP utilizzano un insieme di tecniche, tecnologie, algoritmi che permettono di trattare un segnale continuo successivamente alla sua digitalizzazione.

Le radici dei DSP risalgono agli anni ’60-’70 con l’avvento dei primi computer digitali ed il primo DSP monolitico lanciato sul mercato nell’anno 1978 sotto il nome di Intel 2920.

Questi dispositivi sono classificati a seconda dell’ampiezza e del tipo di dato che sono in grado di elaborare: si parla ad esempio di DSP a 32, 24 oppure 16-bit, a virgola fissa o virgola mobile.

Ogni DSP è quindi adatto ad applicazioni specifiche: ad esempio, i DSP a 16-bit a virgola fissa sono impiegati per il condizionamento di segnali vocali e trovano il loro principale campo di applicazione nella telefonia (fissa e mobile), mentre i DSP a 32-bit in virgola mobile, avendo una dinamica decisamente superiore, sono principalmente impiegati nell'elaborazione di immagini e nella grafica tridimensionale [2].

Conseguentemente a quanto detto, possiamo così convertire i segnali analogici in digitali, elaborare i segnali con il DSP e, se necessario, portare i segnali in uscita di nuovo nel mondo analogico.

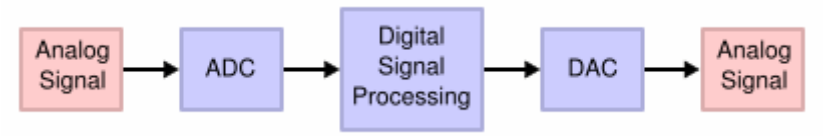


Fig. 1.1 Schema base di elaborazione di un segnale con DSP

È interessante individuare come questi sistemi siano un importante trait d’union tra gli ASIC (Application Specific Integrated Circuits) – che sono circuiti integrati creati appositamente con lo scopo di risolvere un’applicazione di calcolo ben precisa - e i GPP (General Purpose Processors) – dispositivi elettronici che non sono dedicati ad un solo possibile utilizzo - combinando così la velocità dei primi con la convenienza della riprogrammabilità dei secondi.

1.2.2. Programmable Logic Device (PLD)

Nel mondo embedded figura un altro dispositivo che domina una cospicua parte di mercato; questo genere di componente elettronico è conosciuto con l'acronimo di PLD ovvero Programmable Logic Device.

Un PLD è un circuito elettronico utilizzato come parte di circuiti digitali; a differenza di una porta logica, il cui compito è vincolato all'implementazione di una predefinita e non modificabile funzione logica, il dispositivo, al momento della fabbricazione, non è configurato per svolgere una determinata funzione logica.

I primi componenti appartenenti a questa grande famiglia sono stati introdotti sul mercato intorno alla metà degli anni settanta con il nome di PAL (Programmable Array Logic). Essendo una grande famiglia, i membri da cui è costituita non sono pochi; infatti, un po' a causa della legge di Moore e un po' a causa dell'enorme evoluzione delle tecnologie per la realizzazione di circuiti integrati, successivamente sono stati introdotti dispositivi sempre più complessi quali i PLA (Programmable Logic Array), i GAL (Generic Array Logic), i CPLD (Complex Programmable Logic Device), gli FPGA (Field Programmable Gate Array) ed infine gli MPGA (Mask Programmable Gate Array).

Questi dispositivi nella loro architettura fondamentale, sono costituiti da celle ed interconnessioni; principalmente le celle svolgono delle funzioni logiche complesse mentre le interconnessioni permettono di effettuare il collegamento tra le varie celle [1].

Qui di seguito un esempio di architettura di un FPGA.

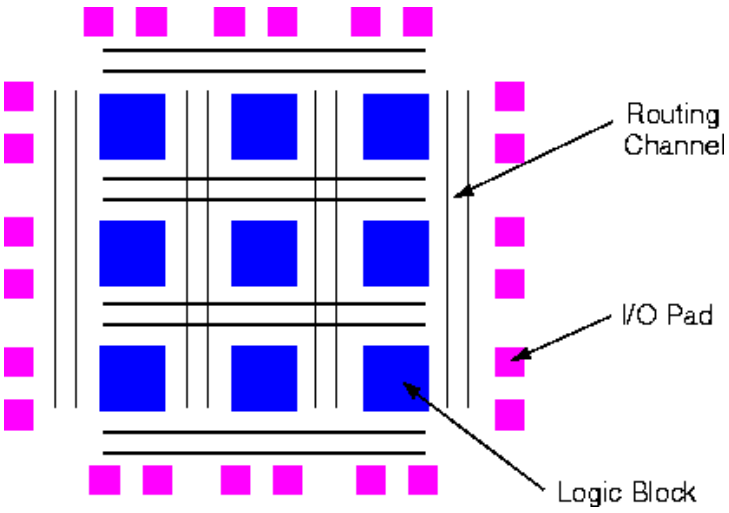


Fig. 1.2 Schema archietturale di un FPGA

Programmare questi componenti sarebbe una questione assai complessa se non fosse che sono stati sviluppati appositi software, chiamati Logic Compiler (un esempio tra tutti il VHDL) attraverso i quali l'utente può gestire ad alto livello le varie caratteristiche da applicare generando un *bitstream*, ovvero flusso di bit, che attraverso un Programmer Hardware, andrà a configurare il dispositivo programmabile.

1.2.3. Microcontrollore (µC)

Oltre all'elaborazione numerica dei segnali, un'altra applicazione di estrema rilevanza nell'ambito dei circuiti integrati, è l'attività di controllo.

Un *microcontrollore* o *microcontroller*, detto anche *computer single chip* è un sistema a microprocessore completo, integrato in un solo chip, progettato per ottenere la massima autosufficienza funzionale ed ottimizzare il rapporto prezzo-prestazioni per una specifica applicazione, a differenza ad esempio, dei microprocessori impiegati nei personal computer (GPP), adatti per un uso più generale.

I µC sono la forma più diffusa e più invisibile di computer. Comprendono la CPU, un certo quantitativo di memoria RAM e memoria ROM (può essere PROM, EPROM, EEPROM o FlashROM) e una serie di interfacce di I/O (input/output) standard, fra cui molto spesso I²C, I²S, SPI, CAN, UART, Ethernet ed altre.

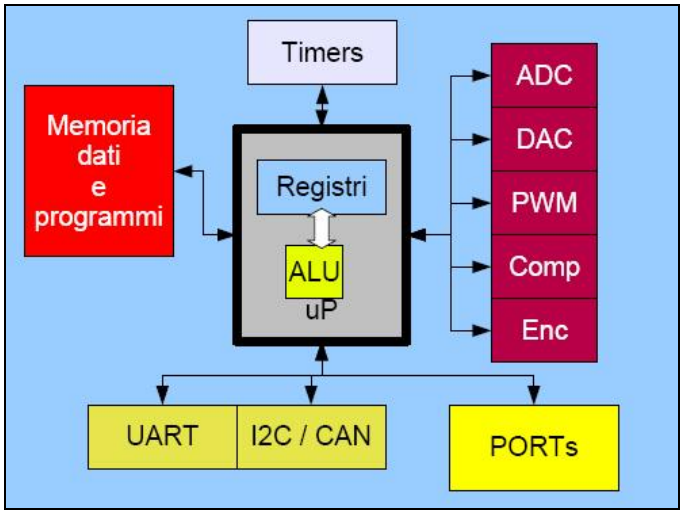


Fig. 1.3 Schema a blocchi di un microcontrollore

Le periferiche integrate sono la vera forza di questi dispositivi: si possono avere convertitori ADC e convertitori DAC multicanale, timer/counters, USART, numerose porte esterne bidirezionali bufferizzate, comparatori, PWM, etc.
I μC sono contenuti in una gran quantità di apparecchi ed elettrodomestici, come ad esempio, videoregistratori e televisori costruiti dopo il 1990, nelle macchine fotografiche e nelle videocamere, nei lettori CD e DVD, nei forni a microonde, nei controlli automatici di macchine industriali, in molte lavatrici e frigoriferi di ultima generazione, nelle centraline di controllo delle motociclette e delle automobili (anche molte decine di μC in una sola automobile), negli antifurto elettronici, nei registratori di cassa dei negozi, negli sportelli Bancomat, nelle centraline dei semafori.
La loro capacità di calcolo è molto limitata, a dispetto della velocità ragguardevole che possono raggiungere e di solito eseguono lo stesso programma (firmware) per tutta la durata del loro funzionamento. In rari casi (difetti molto gravi) il fabbricante del dispositivo che questi controllano può disporre un aggiornamento del firmware.
Negli ultimi anni il loro uso è aumentato grazie all'estrema versatilità ed al costo bassissimo, diffondendosi anche fra gli hobbisti e gli appassionati di elettronica [3].

1.3. Breve cenno sulla componente software

Come un'automobile resta ferma, a causa dell'assenza del suo autista rendendosi così inutile al suo scopo, anche i DSP e i μC, da soli come hardware, non servono a molto.
Infatti, se nel caso della macchina ciò che era necessario a gestire il suo funzionamento era un guidatore, nel caso dei nostri componenti hardware è puramente codice; quest'ultimo è il pilota di turno ovvero è in grado di governare l'architettura e le funzionalità di tali dispositivi.
Questo codice viene principalmente scritto nei linguaggi C o Assembly; il primo è un linguaggio più vicino all'uomo che alla macchina (linguaggio ad alto livello) mentre il secondo è strettamente legato al linguaggio macchina vero e proprio e quindi poco adatto ad una gestione da parte di un essere umano.
Nel caso dei DSP, il codice che viene caricato nella memoria principale del dispositivo è atto ad una gestione efficiente del dispositivo con lo scopo del signal processing.

Altresì nel caso dei microcontrollori (caso che ci interessa direttamente), il software che viene caricato all'interno della memoria principale è, nella stragrande maggioranza dei casi, un Sistema Operativo in Tempo Reale o tecnicamente conosciuto come RTOS.

1.4. Stato dell'Arte

In un campo così vasto come quello dei Sistemi Embedded, non è così semplice poter stabilire quale sia realmente lo stato dell'arte.
Un po' a causa della vastità degli impieghi, un po' a causa del gran numero di software dedicati a tali dispositivi, vi è una continua evoluzione che porta a stabilire solo momentaneamente ed in un brevissimo lasso di tempo, quale sia realmente il sistema per eccellenza.
Per quanto ci riguarda, il dispositivo hardware utilizzato per i nostri scopi sperimentali, è il μC della famiglia ARM7 modello LPC2378 prodotto dalla NXP (founded by Philips); per quanto riguarda il software che verrà caricato su di esso saranno i successivi tests e i gli ulteriori sviluppi a decretare quale possa essere il più adatto alle nostre esigenze.

2. Confronto tra i vari RTOS

2.1. Descrizione di un RTOS

Un Sistema Operativo Real-Time, o in tempo reale (RTOS), è un sistema operativo specializzato per il supporto di applicazioni software Real-Time.
Questi sistemi vengono utilizzati tipicamente in ambito industriale (controllo di processo, pilotaggio di robot, trasferimento di dati nelle telecomunicazioni) o comunque dove sia necessario ottenere una risposta dal sistema in un tempo massimo prefissato. Da un punto di vista puramente teorico l'intervallo di tempo in cui il sistema operativo/applicativo deve reagire non ha importanza, infatti un RTOS non deve essere necessariamente veloce, la cosa importante è che risponda entro un tempo massimo ben conosciuto [4].
Un RTOS deve garantire un' elaborazione rapida dal punto di vista temporale, anche se è possibile che la risposta non sia precisissima. Ad esempio una funzione di calcolo può individuare il peso di un oggetto senza giungere alla precisione del milligrammo poiché deve comunque fornire una risposta in un preciso tempo da quando si è posto il peso nella bilancia. Tale ragionamento non significa che si possano anche dare risposte errate ma che si debba spostare l'attenzione sul tempo della risposta.
Questi tipi di sistemi devono dare l'opportunità allo sviluppatore di conoscere a priori le pessime probabilità in cui si ottiene la risposta.
Viceversa un sistema operativo "tradizionale" deve garantire un'elaborazione corretta dal punto di vista logico, anche se è possibile tollerare che qualche risposta arrivi in anticipo o in ritardo.
I RTOS si possono dividere in due categorie [5]:

- *Sistemi Hard*: richiedono una rigida precisione nella risposta in termini temporali; infatti il mancare una scadenza ha come conseguenza quello di invalidare il funzionamento dell'intero sistema. Un esempio di sistema hard potrebbe essere quello di una catena di montaggio, in cui basta che un pezzo abbia un ritardo e l'intera catena si blocca perché quel pezzo è indispensabile.

- *Sistemi Soft*: si limitano ad un rispetto statistico (tolleranza) dei vincoli di tempo che, qualora prolungati, portano ad un degrado dell'applicazione; degrado che può essere comunque tollerato, in funzione dell'importanza per l'utilizzatore in termini di costo. Sostanzialmente questa distinzione si traduce nella diversa quantificazione dei costi di una possibile inesattezza temporale del sistema.

  Un esempio di soft RTOS può essere un riproduttore DVD, in cui il mancato rispetto dei vincoli si traduce in un degrado della qualità del filmato, ma non pregiudica il proseguimento della riproduzione.

2.2. Parametri di confronto e loro descrizione

Un RTOS può essere descritto come un insieme di servizi di sistema messi a disposizione dello sviluppatore congiuntamente ad un *Task Scheduler* che flessibile e che non richieda troppe risorse sia in termini di occupazione di memoria che di velocità di esecuzione.
Oltre al determinismo e ad un tempo di latenza accettabile (generalmente compreso tra il μs e il ms, ma questo valore può cambiare in modo sostanziale in funzione dell'applicazione) sono svariati gli aspetti che caratterizzano un RTOS.
Esso deve essere *multithread*, ossia deve consentire l'esecuzione di più attività concorrenti e deve disporre di un meccanismo che permetta di assegnare una priorità ereditaria ad ogni *thread* (sottoprocessi). Inoltre deve essere possibile arrestare e riprendere ciascun thread in qualunque istante, compatibilmente con la risoluzione temporale del sistema.
Questa caratteristica, detta *Preemptive* ovvero con diritto di prelazione, permette di rispondere in tempo reale agli eventi critici sospendendo immediatamente i compiti meno importanti a vantaggio di quelli essenziali alla missione del sistema.
Quelle appena descritte sono proprietà che si possono attribuire al nucleo, o *Kernel*, del sistema operativo, ma un buon RTOS deve disporre anche di altre qualità, che in larga parte si riferiscono ad un aspetto che nello sviluppo di sistemi embedded ha una importanza strategica fondamentale: minimizzare il time-to-market.
In generale tra tutte le caratteristiche che indirizzano alla scelta di uno specifico RTOS piuttosto che ad un altro, spiccano alcuni parametri temporali e spaziali che vale la pena di analizzare singolarmente in maniera tale da avere una maggior coscienza nella scelta del sistema giustificata da una più approfondita conoscenza.

2.2.1. File System

Quando si tratta di archiviare in un qualche tipo di memoria una quantità di dati, è necessario mantenere una tracciatura di tale memoria.
In informatica un File System è un meccanismo con il quale i file sono immagazzinati e organizzati su un dispositivo di archiviazione, come un hard disk o un CD-ROM.
Più formalmente, un File System è l'insieme dei tipi di dati astratti necessari per la memorizzazione, l'organizzazione gerarchica, la manipolazione, la navigazione, l'accesso e la lettura dei dati.
Da qui però sorge un problema; a causa della pressante richiesta di esecuzione dei processi entro una rigida deadline, la memoria associata a tali dispositivi embedded, dovendo soddisfare tali margini di tempo, dovrà essere necessariamente rapida ed efficiente in ogni sua operazione dal fetch allo store; risulta quindi molto importante abbinare una memoria di rapida locazione ad un File System in grado di gestirla in maniera adeguata.
I File System possono essere usati su diverse piattaforme che hanno necessità di avere accesso a funzioni hardware di base.
Alcuni esempi tra i più noti dei File System sono i FAT (File Allocation Table) che possono essere del tipo FAT12, FAT16 e FAT32 a seconda di quanti bit si allochino per numerare i *cluster* del disco.
In particolar modo, nel nostro caso, necessitiamo di un File System che sia una libreria estremamente performante ed ottimizzata in maniera tale da soddisfare particolari vincoli su determinati parametri fondamentali tra cui la velocità, la versatilità ed il Footprint.

2.2.2. Footprint

Cercando sul dizionario la parola Footprint, il primo termine che si trova è *impronta*. Subito dopo vi è la parola *ingombro*; ed è infatti proprio d'ingombro che andremo a parlare, o meglio, di occupazione di memoria.
Il Memory Footprint si riferisce all'ammontare di memoria che il programma occupa o a cui si riferisce direttamente mentre è in esecuzione.
Questo ingombro include tutte le regioni di memoria attive come la regione contenente il codice, le sezioni di dati statici, lo *Heap*, lo *Stack* e tutte le altre; chiaramente più il programma è ingombrante, più il Memory Footprint avrà una dimensione superiore.

Nel caso in cui le dimensioni del sistema ed il tempo di esecuzione siano predominanti, come ad esempio nei Sistemi Embedded, avere un basso Footprint di memoria risulta essenziale.
In tal senso, è bene fare distinzione tra i tipi di memoria con cui si andrà ad avere a che fare sulla piattaforma hardware di proprio interesse; quasi nella totalità dei casi la memoria si divide in memoria ROM (Read Only Memory) e memoria RAM (Random Access Memory).
Nella memoria ROM è consentita solo l'operazione di lettura, ed è appunto per questo che viene utilizzata per immagazzinare definitivamente il programma che poi si andrà ad eseguire sulla propria piattaforma; negli ultimi tempi questo tipo di memoria è stato sostituito dalle Flash Memory che sono memorie permanenti riscrivibili, tecnicamente conosciute come EEPROM (Electrically Erasable and Programmable Read Only Memory), organizzate a blocchi.
Per quanto riguarda la memoria RAM, a differenza della precedente, è il supporto di memoria su cui è possibile scrivere e leggere informazioni con un accesso di tipo "casuale", ovvero senza dover rispettare un determinato ordine; questa solitamente, nei Sistemi Embedded, ha dimensioni minori della memoria di programma vista sopra.
Risulta quindi essere un requisito fondamentale la conoscenza indicativa a priori del Footprint di un dato sistema RTOS in maniera tale da poter riconoscere subito l'eventuale incompatibilità tra sistema e piattaforma hardware.

2.2.3. Context Switch Time (CST)

Nella valutazione dei sistemi RTOS uno dei più significativi parametri su cui basare la propria scelta è il tempo di commutazione di contesto, meglio conosciuto come *Context Switch Time* (CST); è dunque utile nei sistemi con un solo processore (come nel nostro caso) per effettuare un "apparente parallelismo" di calcolo.
Questo tempo è determinato da quella parte del sistema operativo che fa commutare da un processo ad un altro ovvero lo *scheduler*; infatti, come vedremo nel paragrafo successivo, il CST ha una varianza molto spiccata in diretta relazione all'algoritmo che verrà implementato per realizzare lo scheduler.
Il CST è esattamente il tempo che impiega un RTOS per passare allo stato di esecuzione da un processo ad un altro; questo tempo è causato da diversi fattori tra cui il salvataggio ed il

caricamento dei registri e delle mappe di memoria; anche in questo caso ci sono degli inconvenienti che possono essere ovviati con degli adeguati accorgimenti relativi al CST come possono essere un adeguato progetto o un'adeguata implementazione dell'algoritmo di scheduling [6].

2.2.4. Interrupt Latency

Quando si passa da un processo ad un altro le cause possono essere principalmente due, entrambe legate allo scheduling.
Come visto nel paragrafo precedente, quando la commutazione di processo è provocata dall'attuarsi dell'algoritmo di scheduling, si parla di CST. Spesso, però, nei sistemi Real-Time, la richiesta di avviare un nuovo processo proviene da un fattore esterno ovvero da una periferica che effettua una richiesta di eseguire immediatamente il task a lei legato in quanto urgente; questa richiesta in tempo reale prende il nome di *Interrupt Request (IRQ).*
In questo caso, il tempo che intercorre tra la richiesta di interrupt e l'inizio dell'esecuzione del processo relativo a tale interruzione, è detto *Interrupt Latency*.

2.2.5. Scheduler

Ognuno di noi si sarà certamente reso conto di come sia problematico effettuare più cose contemporaneamente; da qui nasce la necessità di eseguire una cosa alla volta, seguendo un determinato criterio, volto a non perderne alcuna di vista; tale criterio è l'*algoritmo di scheduling.*
Nei sistemi operativi *multitasking*, cioè quelli in grado di eseguire più processi (*task*) concorrentemente, è presente lo *scheduler*, componente fondamentale atto a far avanzare un processo interrompendone temporaneamente un altro, realizzando così un cambio di contesto (Context Switch).
Uno scheduler è completamente determinato dall'algoritmo che esso implementa; la varietà dei diversi algoritmi di scheduling permette di scegliere, nella maniera più efficiente possibile, le operazioni da effettuare sui vari tasks.
Introduciamo alcuni concetti e definizioni di base che caratterizzano lo studio della teoria e degli algoritmi della schedulazione Real-Time. Definiamo anzitutto il concetto di processo.
Per *processo* o *task* intendiamo una sequenza di istruzioni che, in assenza di altre attività, viene eseguita dal processore in modo continuativo fino al suo completamento.

La situazione di riferimento per un ambiente Real-Time è quella in cui sono presenti numerosi processi, ciascuno con particolari esigenze di schedulazione. In questo caso è necessario individuare una strategia di assegnazione del processore che sequenzializzi l'uso della risorsa fisica secondo un criterio stabilito a priori.
Gli stati in cui un task vive all'interno dello scheduler sono principalmente tre:

- **RUN:** caso in cui il task è attualmente in esecuzione.
- **READY:** i tasks che appartengono a questa categoria sono in grado di essere lanciati in esecuzione ma non sono correntemente in esecuzione poiché un differente task di uguale o maggiore priorità è già nello stato di run.
- **STOP:** un task è in tale stato se sta correntemente attendendo un evento temporale (un timer) o un evento esterno (un semaforo). I tasks in questo stato non sono disponibili allo scheduling.

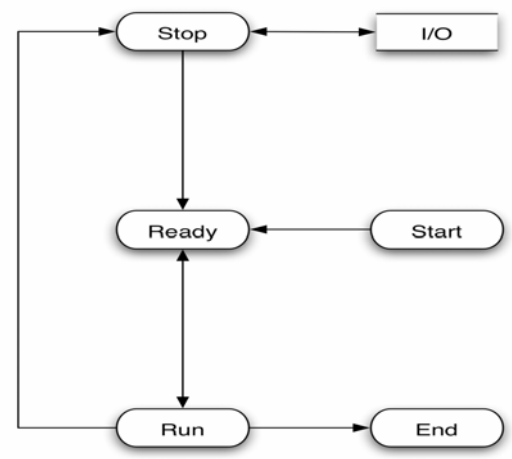


Fig. 2.1 Stato dei processi di uno scheduler generico

Quanto visto sopra è soltanto uno schema generale di ciò che deve, al minimo, effettuare uno scheduler; infatti nei RTOS buona parte dell'esecuzione dei processi è richiesta da un interrupt. A questo punto, stabilito per sommi capi il compito dello scheduler è di indubbio interesse analizzare le logiche e le motivazioni che spingono a scegliere un dato algoritmo di scheduling piuttosto che un altro; l'implementazione di tale algoritmo è proprio lo scheduler [7].
Chiaramente chi comanda è l'applicazione e ad ogni applicazione corrisponde un appropriato algoritmo di scheduling infatti è lapalissiano che, controllare un reattore

nucleare piuttosto che un trenino elettrico in un plastico, richieda un diverso ordinamento delle priorità ed una logica di gestione dei tempi di vita di processo molto differenti tra loro.
Risulta dunque interessante analizzare in breve i principali algoritmi maggiormente implementati nei sistemi Real-Time, o più precisamente nei RTOS, con lo scopo di realizzarne lo scheduler.
Ci sono numerosi approcci per "schedulare" i processi nei sistemi Real-Time; questi ricadono principalmente in due categorie:

- Schedulazione a priorità statica
- Schedulazione a priorità dinamica

Numerosi RTOS contemporanei utilizzano la politica a priorità statica secondo la quale non è previsto alcun cambiamento nelle priorità di lavoro mentre il processo è in stato di RUN. Questo approccio richiede un supporto a livello di codice piuttosto leggero per implementare le sue funzionalità; in questo modo lo scheduler è veloce ed effettua le operazioni in maniera deterministica. È inoltre interessante notare che lo scheduling è principalmente definito in anticipo a tempo di compilazione (*compile-time*).
Direttamente contrapposto troviamo il secondo ovvero lo scheduler dinamico; esso permette di modificare la priorità di lavoro basata su diverse politiche di algoritmi di schedulazione. Questo approccio è più complicato e richiede più codice per implementare tale logica oltre a necessitare di un notevole *overhead* per gestire i processi.
È invece qui prevista una logica non deterministica la quale è piuttosto sfavorevole per i sistemi hard Real-Time. In questo modo il task attivo in un dato momento, può modificare il suo assetto mentre è in fase di RUN e quindi la priorità dei processi può cambiare dinamicamente.
È stato particolarmente ostico riuscire ad individuare quali fossero degli algoritmi effettivamente più usati di altri; nella ricerca sulla rete abbiamo notato che è presente una vastissima quantità di sistemi RTOS ed ognuno di questi, per adattarsi a specifiche applicazioni e piattaforme hardware, ha caratteristiche ben discostate dai suoi ipotetici rivali di mercato.

Nonostante ciò abbiamo individuato alcuni fili conduttori legati a specifici tipi di algoritmi di scheduling e li abbiamo così analizzati più a fondo in maniera tale da offrire un'infarinatura di quali fossero le principali caratteristiche e peculiarità di un algoritmo di schedulazione; vediamo dunque alcuni esempi tra gli algoritmi di scheduling maggiormente implementati.

2.2.5.1. Preemptive scheduling

Il preemptive scheduler (con diritto di prelazione) è uno dei più utilizzati nei sistemi RTOS. Nella stragrande maggioranza dei casi si usa uno scheduler *fixed priority preemptive* ovvero uno scheduler che assicura in ogni momento che il processore esegua il task con la priorità più alta (fissata precedentemente dal programmatore) tra tutti i tasks che in quel momento sono nello stato di READY.
Il suo funzionamento è piuttosto semplice infatti questo tipo di scheduler è governato da un timer, la cui durata predefinita è denominata *quantum of tick;* trascorsa una di queste unità di tempo, viene effettuata una cernita tra i processi in stato di READY e quello con maggiore priorità viene portato in stato di RUN ed eseguito almeno per un quanto del timer.

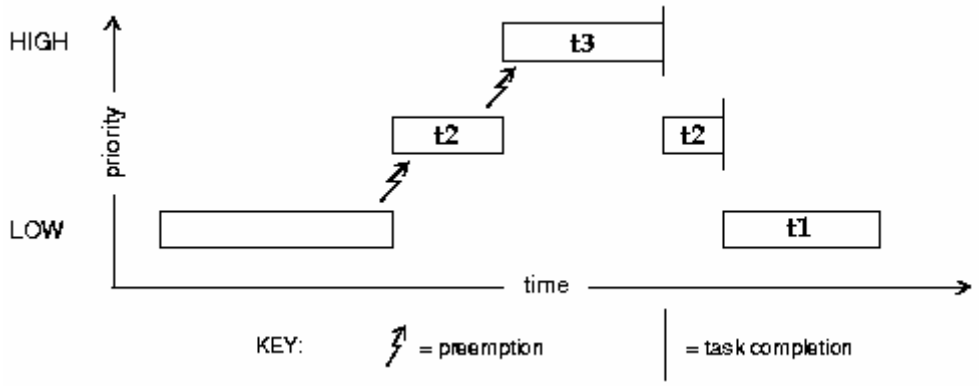


Fig. 2.2 Diagramma temporale di un preemptive scheduler

Questo sistema di schedulazione ha il vantaggio di evitare un monopolio del processore da parte di un task per un tempo maggiore del countdown del timer.
Come ogni medaglia, però, anche questa ha il suo rovescio che prende il nome di *Starvation* [8]; tale fenomeno si verifica quando, un processo pronto all'esecuzione, è impossibilitato ad ottenere le riscorse di cui necessita. Nel nostro caso può succedere che

un task con priorità molto bassa venga sottomesso al sistema da processi con priorità più alta quali sono ad esempio gli interrupt.
Per ovviare a questo problema, oltre che a poter utilizzare degli algoritmi di scheduling diversi, si utilizzano le cosiddette tecniche di invecchiamento (*aging*) ovvero si provvede ad aumentare progressivamente, ad intervalli regolari, la priorità del processo sottomesso.
Inoltre questo tipo di schedulazione, avendo un quantum of tick costante e frequente che comporta una continua commutazione di contesto, sarà di estrema importanza avere un RTOS il cui scheduler abbia un CST molto basso.

2.2.5.2. Non-preemptive scheduling

Togliendo il timer si ottiene, a partire dal precedente, questo algoritmo di scheduling.
Se prima era il timer a decidere quando fosse il momento di commutare da un processo ad un altro, ora il task, in stato di RUN, deve essere portato fino al termine dell'esecuzione.
Gli algoritmi di schedulazione non-preemptive, sono progettati in maniera tale che ogni processo in esecuzione rimanga tale sino a quando volontariamente invochi servizi del sistema operativo.
Si ha così una notevole semplificazione nei rapporti tra processi ed inoltre viene minimizzato l'overhead garantendo un miglior sfruttamento delle risorse.
Il grosso limite di questo approccio è che le prestazioni Real-Time sono responsabilità del programmatore di processi a tempo di compilazione; le prestazioni sono, quindi, per nulla garantite dal sistema operativo.
In questo caso c'è da notare che un CST estremamente ottimizzato non è effettivamente necessario in quanto la commutazione di contesto è molto meno frequente rispetto al caso precedente.

2.2.5.3. Round-Robin scheduling (RR)

L'espressione Round-Robin viene usata in molti contesti per riferirsi ad un sistema in cui diversi partecipanti ad una attività si alternano in modo circolare.
Infatti questo nome deriva dall'espressione francese in uso nel diciassettesimo secolo *ruban rond* che significa fiocco rosso; essa veniva usata per riferirsi ad un modo di disporre in circolo le firme di una petizione inviata alle autorità in modo che non fosse possibile identificare un capolista di quest'ultima o un ordine gerarchico tra i firmanti.

Nello scheduling RR si fissa un intervallo di tempo detto *time-slice* (quantum of tick) e si assegna il processore ad ogni task nel pool a rotazione per una quantità di tempo pari alla *slice*.

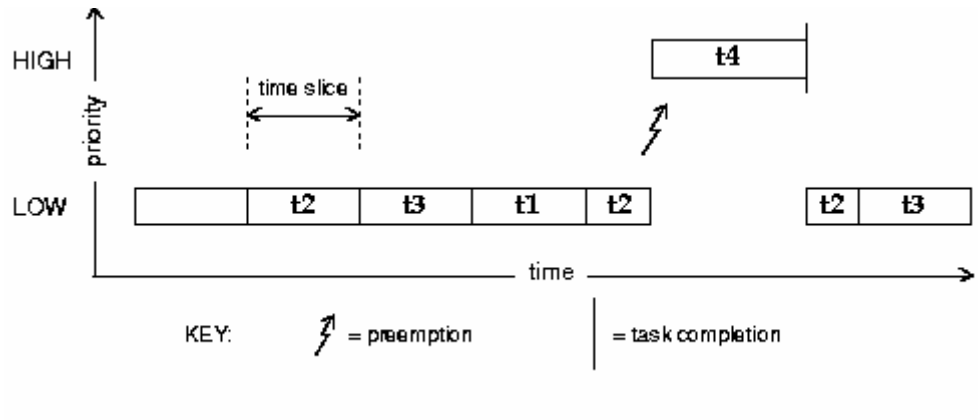


Fig. 2.3 Diagramma temporale di un Round-Robin scheduler

Questo tipo di scheduling consente di simulare l'esecuzione contemporanea di più processi; infatti, se la slice di tempo è molto piccola, si può avere l'impressione che il processore esegua contemporaneamente più tasks anche se ad ogni istante di tempo il processore può eseguire al più un processo.
Quando un task viene inserito nel pool, lo scheduler lo mette in fondo alla coda e gli assegna il processore solamente dopo averlo assegnato una volta a tutti gli altri processi.
In questo caso i tasks possono essere sospesi durante la loro esecuzione e quindi lo scheduler è da considerarsi con prelazione. Tuttavia, lo scheduling viene eseguito ad intervalli regolari che non dipendono da quando viene inserito un nuovo processo nel pool.
Le prestazioni di questo algoritmo sono dunque influenzate dal tempo medio di attesa sebbene consenta a tutti i processi di ottenere il controllo del processore ed eviti quindi il problema dell'attesa indefinita vista in precedenza (Starvation).
In questo algoritmo di scheduling il CST assume un'importanza estremamente rilevante in quanto la sua potenza deriva proprio dal continuo cambio di contesto nei processi.

2.2.5.4. FCFS scheduling

L'algoritmo di schedulazione FCFS (First Come First Served) è basato sulla logica FIFO (First In First Out); esso infatti esegue i processi nello stesso ordine in cui essi vengono forniti al sistema. Il primo processo ad essere eseguito è esattamente quello che per primo

richiede l'uso del processore. Quelli successivi vengono serviti non appena questo ha terminato la propria esecuzione, e così avviene di seguito per tutti gli altri posti in coda.
Questo tipo di algoritmo è molto semplice da implementare ma solitamente è anche poco efficiente, almeno considerando il tempo medio d'attesa.
L'FCFS è un algoritmo senza diritto di prelazione ed ha prestazioni migliori con processi lunghi mentre penalizza i processi brevi.

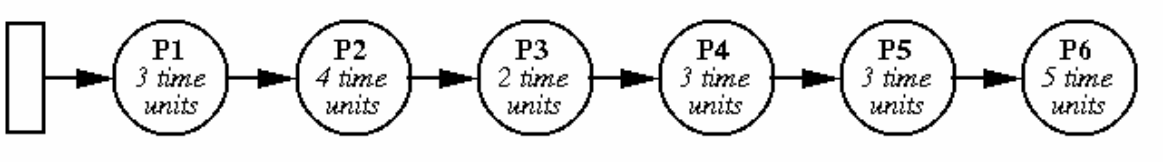


Fig. 2.4 Sequenza temporale di un FCFS scheduler

I processi schedulati con una logica algoritmica di questo genere, al contrario dei tipi con diritto di prelazione, non soffrono del fenomeno della Starvation.
Come nel caso dello scheduler non-preemptive, anche in questo caso non è necessario ottimizzare il CST poiché la frequenza di cambio di contesto è talmente bassa che il CST non va ad intaccare, con una percentuale confrontabile, il totale del tempo.

2.2.6. Supported Platforms

Ad ogni stagione, il giusto abbigliamento. Ogni RTOS, nel proprio datasheet, ha un elenco di piattaforme hardware sulle quali esso può essere "fatto girare".
Quando si ha da risolvere un problema, come nel nostro caso, di controllo, è necessario effettuare una ricerca per vedere quale possa essere, in base alle sue caratteristiche, una piattaforma hardware adeguata (i.e. ARM7/9, Cortex M3, AVR, PIC ed altri).
Per piattaforma adeguata si intende un hardware, come può essere ad esempio un μC, che sia ottimizzato ed abbia una valida struttura hardware affinché l'applicazione che andrà ad essere portata (caricata) venga eseguita nel modo migliore.
Generalmente, una volta scelta la piattaforma, si passa direttamente alla ricerca di un RTOS che sia stato progettato, o perlomeno facilmente portabile (adattabile), in maniera tale da sfruttare, attraverso un'attenta ottimizzazione, la piattaforma hardware sulla quale verrà caricato cosicché si possa raggiungere un opportuno valore di efficienza; non sempre la prassi è così poiché spesso entrano in gioco fattori che possono mutare tale linea guida.

2.3. Cenni dei RTOS confrontati

Siamo giunti qui ad uno dei punti cruciali della trattazione; saranno ora analizzati, nei parametri a noi utili, sette RTOS che sono stati scelti, tra i tanti disponibili sul mercato elettronico, come candidati a diventare il "nostro" RTOS.

A proposito di mercato, c'è una grossa distinzione da fare; innanzitutto esistono RTOS con licenza proprietaria e licenza open-source (software rilasciato con un tipo di licenza per la quale il codice sorgente è lasciato alla disponibilità di eventuali sviluppatori in modo che, con la collaborazione, il prodotto finale possa raggiungere una complessità maggiore di quanto potrebbe ottenere un singolo gruppo di programmazione). I primi devono essere acquistati necessariamente del rivenditore autorizzato o direttamente dall'azienda sviluppatrice, i secondi, invece, sono liberamente scaricabili dalla rete e liberamente modificabili e ottimizzabili.

Poiché buona parte delle informazioni di nostro interesse non era disponibile direttamente in rete, abbiamo inviato, ad ogni distributore, una e-mail in cui si chiedevano ulteriori chiarimenti sulle prestazioni dei relativi sistemi proprietari.

Uno dei più, se non il più, conosciuto RTOS con licenza open-source è **FreeRTOS**: esso è progettato per piccoli μC ed è scritto in linguaggio C, ha un scheduler di tipo preemptive e i threads in coda, che condividono lo stesso livello di prelazione, sono schedulati in modalità Round-Robin. Il File System supportato può essere di tipo FAT16/32 e ha un footprint estremamente ridotto [9].

Tutti gli altri RTOS analizzati hanno licenza proprietaria; questo però non deve scoraggiare al loro utilizzo in quanto, alcuni di essi, hanno caratteristiche talmente performanti da portare in secondo piano il costo come metrica di scelta.

Il primo RTOS con licenza proprietaria che siamo andati ad analizzare è **RTXC Quadros**: esso è un sistema che si presta con buoni risultati al general processing e ad applicazioni di controllo, supporta File System di tipo FAT12/16/32, ha uno scheduler preemptive e, nel caso ci siano più processi a stessa priorità, adotta una politica di distribuzione RR.

Ha un footprint sufficientemente piccolo tale da renderlo adatto ad applicazioni che richiedono un'alta frequenza di interrupt e un CST discretamente ridotto [10].

Il secondo sistema analizzato è **Segger EmbOS**: il suo grosso vantaggio è quello di essere progettato per offrire una reale sensazione di multitasking, ha un footprint piuttosto ridotto,

ha anch'esso uno scheduler come quello dei due sistemi precedenti, un'interrupt latency e un Context Switch Time piuttosto contenuti [11].

Il terzo RTOS da noi preso in considerazione è **Express Logic ThreadX**: esso è stato progettato specificatamente per applicazioni embedded, è implementato come libreria C e ha un footprint ragionevolmente ridotto. È dotato di una gestione degli interrupt ottimizzata, un Context Switch che offre una buona reattività e supporta un File System FAT-compatibile e offre uno scheduler basato sulla stessa logica di quelli precedentemente analizzati [12].

Il quarto sistema operativo analizzato è **QNX Neutrino**: esso è dotato di un microkernel il quale permette di essere caricato su dispositivi molto piccoli mostrando così un footprint molto ridotto. Il sistema mette a disposizione una serie di opzioni per supportare diversi File System, ha un efficiente gestione del Context Switch, un'interrupt latency piuttosto ridotta e implementa diversi tipi di algoritmi di scheduling tra cui FCFS e RR [13].

Il quinto sistema passato in analisi secondo i medesimi parametri dei RTOS precedenti è **LINUXWORKS LynxOS**: esso è stato progettato per applicazioni Hard Real-Time ed è dotato di una scalabilità lineare che ne fa incrementare la predicibilità operativa. Questo sistema è stato progettato in maniera modulare così da rendere meglio gestibile la quantificazione del footprint. Supporta diversi dispositivi di interrupt ai quali risponde in un tempo ragionevolmente accettabile. Questo sistema è dotato di un Context Switch deterministico grazie ad uno scheduler a rapida gestione.

Per quanto riguarda lo scheduler questo sistema ne presenta quattro tra i più comuni ed utilizzati e supporta diversi File System tra cui Lynx Fast File System e ISO 9660 File System; purtroppo però non sembra supportare il File System di tipo FAT. Anche questo sistema, malgrado le buone caratteristiche operative ha licenza proprietaria [14].

Sesto ed ultimo RTOS con licenza proprietaria analizzato è lo **IAR PowerPac**: innanzi tutto è dotato di un footprint molto ridotto; nonostante ciò diverse caratteristiche del sistema sono già incluse nella versione base. Per quanto riguarda gli interrupt il sistema è totalmente interrompibile in un tempo brevissimo e, questa caratteristica, fa di PowerPac un valido RTOS da utilizzare nelle situazioni time-critical. Gode inoltre di un Context Switch Time estremamente breve che abbinato ad un interrupt latency altrettanto ridotto (Zero Interrupt Latency) crea una responsività del sistema molto interessante. Per quanto riguarda il File System è supportato il FAT nelle sue diverse varianti.

A livello di scheduler vengono offerti un preemptive scheduling nel caso generale e successivamente, qualora si presentino processi con identica priorità, la politica di schedulazione viene commutata in RR. Relativamente alla licenza, anche in questo caso, ci troviamo di fronte ad un sistema con licenza proprietaria [15].

2.4. Scelta dei RTOS di interesse

A questo punto, dopo aver messo a confronto tali sistemi, risulta necessario e doveroso, per quanto ci riguarda, scegliere tra questi i RTOS che più si addicano alle nostre richieste; segue quindi, in base alle informazioni ed ai parametri raccolti una riduzione del campo di scelta, circoscritto a quei sistemi che possano essere i reali candidati a soddisfare le nostre aspettative prestazionali.

Tutti i sistemi analizzati hanno, come si può notare dalla loro descrizione generale, caratteristiche molto simili ma differiscono per certi aspetti che per noi fanno la differenza.

Alla foce di tutto questo ragionamento i sistemi individuati come interessanti per quel che concerne il nostro scopo sono fondamentalmente due: **FreeRTOS** e **IAR PowerPac**; seguono le motivazioni che ci hanno spinto ad affermare ciò e, nei capitoli successivi, andremo progressivamente più a fondo sia operativamente che quantitativamente su questi due sistemi decretando oggettivamente quale sia verosimilmente il più adatto a noi.

Fig. 2.5 Logo di FreeRTOS

Nel primo caso, ovvero per **FreeRTOS** che dire, il pezzo forte di questo sistema operativo risiede nel fatto che offre una licenza Open-Source; questo significa che il codice sorgente è liberamente prelevabile dalla rete e modificabile a proprio piacimento. Inoltre è facile trovare informazioni in rete su come compilarlo e su come effettuare operazioni di porting relativamente ad un ragguardevole numero di piattaforme hardware. Resta da non sottovalutare che questo sistema operativo è in grado di gestire nativamente un buon numero di stack.

Purtroppo, per questo sistema non sono disponibili valori specifici che individuino velocità o caratteristiche di risposta; ma questo non ci scoraggia, infatti, nella fase successiva di questo documento saranno ricercati sufficienti valori attraverso analisi sperimentali in maniera tale da poter così valutare più accuratamente e specificamente le potenzialità di tale sistema. Quindi fondamentalmente il sistema potrebbe anche essere quello adatto,

restano da valutare alcuni parametri che possono confermare o contraddire quanto detto; se ne riparlerà dopo l'analisi sperimentale che avverrà successivamente.

Fig. 2.6 Logo di IAR PowerPac

Per quanto riguarda invece **IAR PowerPac**, partiamo subito con il rovescio della medaglia, ovvero, il sistema ha licenza proprietaria; il suo prezzo si aggira intorno alle decine di migliaia di Euro e, nel caso venga acquistato precompilato, esso è decisamente inferiore; inoltre il supporto in rete è parzialmente limitato a singoli esempi spesso forniti dallo sviluppatore del μC stesso o direttamente all'interno dell'ambiente di sviluppo.

Come però accennato precedentemente, questo fatto non deve scoraggiare l'eventuale acquirente in quanto, se una licenza è supportata, o meglio, giustificata da un insieme di prestazioni considerevoli e da un'assistenza specializzata fornita direttamente da personale in stretto contatto con gli sviluppatori del RTOS stesso, il denaro investito nell'acquisto del prodotto può essere rapidamente ammortato dai risultati ottenuti nel lancio sul mercato. D'altronde non si può costruire una Lamborghini in garage. E proprio a proposito di velocità è bene ragionare su come le prestazioni di questo sistema ci abbiano spinto a farlo entrare nella coppia di sistemi favoriti a diventare l'oggetto di utilizzo quotidiano. Innanzitutto PowerPac è un sistema adatto ad applicazioni *mission-critical* ovvero dove la reazione immediata è fondamentale; questo è dimostrato dalla presenza di una Zero Interrupt Latency e accentuato da un tempo di commutazione di contesto di ordine molto piccolo; anche questo S.O. supporta nativamente una serie di stack particolarmente ottimizzati ed adatti a varie tipologie di microcontrollore.

Insieme a questo RTOS, inoltre, viene fornito anche un ambiente di sviluppo estremamente specializzato ed integrato così da rendere le modifiche implementative più rapide ed efficaci. Sulla base di queste motivazioni principali, possiamo asserire che anche IAR PowerPac, a causa delle sue peculiarità, è candidato a diventare il RTOS da noi utilizzato.

Concludendo questa parte anticipiamo che, nei capitoli successivi, verranno effettuate ulteriori valutazioni sperimentali e test parametrici mirati al termine dei quali verrà scelto tra FreeRTOS e IAR PowerPac "*il sistema*".

3. Strumenti di testing e sviluppo utilizzati

Se per montare un motore di una automobile ci vogliono chiavi inglesi, brugole, banchi di riscontro, etc, anche nel nostro caso necessitiamo di "attrezzi del mestiere" specifici.

Andremo dunque qui di seguito a mostrare una breve, ma esaustiva, rassegna di tutti i componenti hardware e software - la cui fruizione ci è stata gentilmente concessa da ELSAG Datamat - che si sono resi utili, durante le fasi sperimentali, all'acquisizione dei valori relativi ai parametri di confronto dei due RTOS.

La sequenza di test prevede l'utilizzo di diversi strumenti; in primo luogo abbiamo utilizzato un PC Laptop sul quale è stato installato ed adeguatamente configurato l'ambiente di sviluppo permettendoci la completa gestione della parte software e della relativa componentistica d'interfacciamento.

Un altro componente fondamentale è stato il JTAG che ci ha permesso di praticare un'attività di debug sfruttando le sue potenzialità di emulazione; in stretta correlazione funzionale a quest'ultimo vi è l'*Evaluation Board* (EB) facente parte dello *Starter Kit* (STK) che ci ha permesso di accedere alle funzionalità del microcontrollore attraverso la disponibilità di numerose interfacce progettate ad hoc per le valutazioni sperimentali.

Il componente finale che ha permesso di interfacciarci con i risultati dei test è stato l'oscilloscopio che si è rivelato un insostituibile mezzo per la raccolta dei dati temporali.

I suddetti componenti, tranne l'ambiente di sviluppo che è propriamente installato sul Laptop, sono collegati tra loro come visibile in figura:

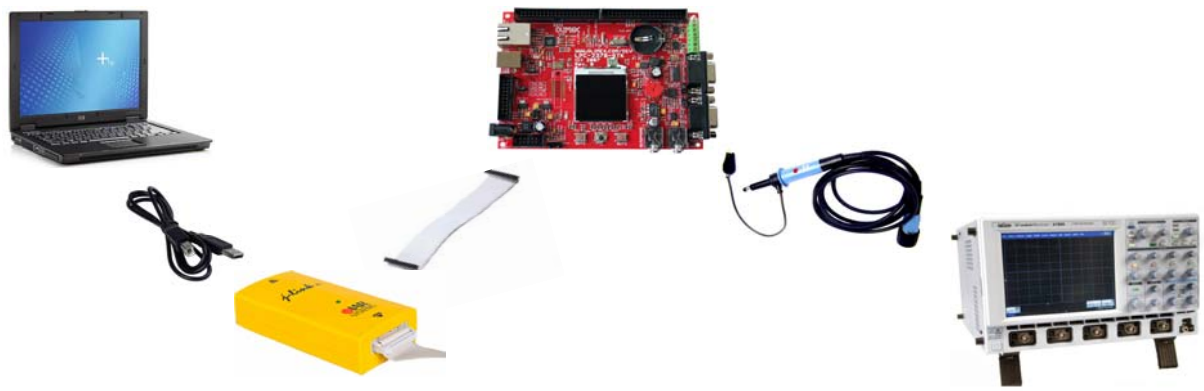


Fig. 3.1 Schema strutturale della sequenza di test

Come visto in figura il tutto può essere riassunto da un insieme di oggetti interconnessi tra loro; andremo ora quindi ad analizzare le particolarità strutturali e funzionali di ciascuno di questi oggetti e, per quanto riguarda le interconnessioni, ci limiteremo a descriverne brevemente l'utilità dovendo quest'ultime adempiere ad un mero compito di trasmissione.

3.1. Descrizione del PC

Fig. 3.2 PC Laptop utilizzato

Il primo strumento di cui ci siamo serviti è stato un Notebook HP nc6320 provvisto di un microprocessore Intel Core 2 Duo T5600 operante ad una frequenza di clock pari a 1.83GHz e con 2GByte di memoria RAM; a bordo di questo Laptop è stato installato come sistema operativo Microsoft Windows XP Professional con Service Pack 2. Per quanto riguarda gli altri componenti hardware on board non è di fondamentale importanza fornirne ora la descrizione specifica in quanto ciò che importa è la loro presenza piuttosto che la loro performance; è sufficiente, quindi, spendere due parole sul fatto che tale Laptop disponga di una scheda madre in grado di fornire interfacce utili ai nostri scopi quali, in primis, l'interfaccia USB.

Oltre che per poterci interfacciare con periferiche atte alle attività di test, tale PC è stato il punto di partenza per tutte le successive attività sperimentali in quanto, grazie all'installazione su di esso dell'ambiente di sviluppo, è stato possibile leggere e modificare i progetti contenenti il codice sorgente del sistema operativo.

3.1.1. Ambienti di sviluppo

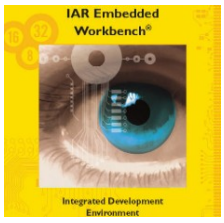


Fig. 3.3 IDE di IAR

A questo punto è doveroso dedicare un intero paragrafo agli ambienti di sviluppo; prima di tutto è bene individuare quali siano le varie componenti, anche se a volte non tutte presenti, caratterizzanti un ambiente di sviluppo specificando in seguito quali siano le loro peculiarità così da rendere più comprensibile il suo significato:

- L'IDE ovvero l'Integrated Development Environment è un ambiente integrato di sviluppo e cioè un editor grafico di codice sorgente che permette di integrare gli

accessi al software per il debug e l'utilizzo di compilatori in maniera implicita. Questo aiuta i programmatori nello sviluppo di applicazioni ed inoltre l'IDE è una costante di sviluppo e non è dipendente dal microcontrollore usato.

- Il compilatore che è un programma che traduce una serie di istruzioni da un determinato linguaggio più vicino all'uomo (codice sorgente) in una serie di istruzioni più vicine alla macchina (codice oggetto).
- Gli strumenti di *building* dove viene attuato il processo di *linking* (da parte del *linker*) che è il procedimento per cui vengono presi uno o più oggetti generati dal compilatore ed assemblati in un unico programma eseguibile.
- Gli strumenti di debug che dipendono dal microcontrollore utilizzato e sono utili all'individuazione di porzioni di software affette da errori (*bugs*) a seguito dell'utilizzo del programma piuttosto che in fase di compilazione.

È bene notare come gli ultimi tre componenti all'elenco precedente, nella maggior parte dei casi, costituiscano la *toolchain* che è direttamente dipendente, a differenza dell'IDE, dal tipo di μC utilizzato; tale catena viene seguita in modo sequenziale durante le attività di sviluppo e test.
Questi ambienti di sviluppo si possono distinguere sulla base delle loro componenti e possono essere mono o multi-linguaggio, orientati agli oggetti o meno, grafici o con interfacce più complesse (i.e. a linea di comando). Qui di seguito vediamo una breve rassegna di alcuni ambienti di sviluppo relativamente alle sopraccitate caratteristiche:

- IAR Embedded Workbench, Keil Embedded Development Tools, eMbedded Visual C++ sono ambienti di sviluppo completi di compilatori specifici per particolari microcontrollori definiti già in fase di acquisto.
- Eclipse è un ambiente di sviluppo che fornisce un ottimo IDE ma che non dispone di una toolchain; questo è sia un vantaggio che uno svantaggio in quanto è possibile utilizzarlo su piattaforme differenti ma necessita della completa installazione di quest'ultima.

Durante la scelta dell'ambiente di sviluppo vanno ricercate le caratteristiche che meglio soddisfino le esigenze dell'applicazione software che si sta sviluppando e che permettano,

ai programmatori, la più confortevole attività di sviluppo integrando, quindi, una toolchain ben fornita ed un IDE particolarmente funzionale.

Da qui segue immediatamente la scelta dell'ambiente di sviluppo più adatto che, nel nostro caso dopo le dovute valutazioni, è risultato essere IAR Embedded Workbench 5.0.

Questo ambiente di sviluppo fornisce una Toolchain per microcontrollori LPCxxxx (ovvero per tutta la famiglia degli LPC) e raggiunge un buon livello di integrazione con gli strumenti per la compilazione, il building automatico ed il debug garantendo, comunque, delle ottime performance in rapporto al prezzo d'acquisto; da notare inoltre come il compilatore da noi utilizzato sia considerato, in letteratura, uno dei migliori per quanto concerne il rapporto Footprint/prestazioni.

Scelto dunque l'ambiente di sviluppo si è installato il pacchetto software sul portatile. L'operazione ha prodotto la copia delle componenti software sul PC e successivamente configurato l'ambiente per essere eseguito sul computer stesso.

All'esecuzione dell'applicazione, sullo schermo, si è presentata la seguente videata:

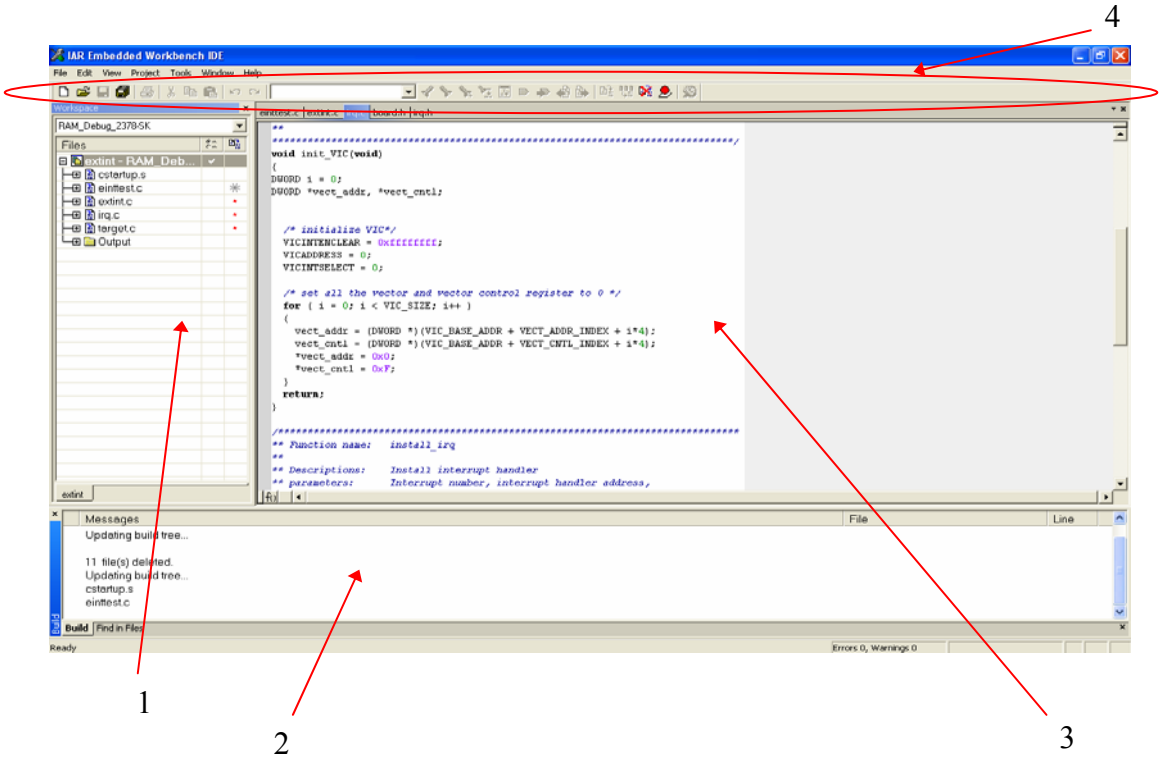


Fig. 3.4 Screenshot dell'IDE IAR Embedded Workbench 5.0

Come si può vedere dallo screenshot in figura relativo all'ambiente di sviluppo, si può notare come l'area di lavoro sia fondamentalmente suddivisa in quattro zone principali che abbiamo numerato per poterle meglio analizzare.

La zona 1 rappresenta la quella di esplorazione; di norma presenta un diagramma ad albero del Workspace relativo al progetto sul quale si sta lavorando ed individua la zona nella quale è possibile modificare i connotati del progetto stesso a livello di file. Tale zona può essere arricchita di ulteriori visualizzazioni qualora l'ambiente sia integrato con programmi di versionamento o altro.

La zona 2 è molto variabile in quanto ciò che risiede in questo modulo è direttamente connesso e dipendente dalle informazioni a cui siamo interessati quali log di compilazione, informazioni di debug, risultati delle ricerche, etc; tra tutte queste possibili caselle di comunicazione all'interno di tale modulo, ha un'importanza rilevante il Terminal I/O il quale permette di visualizzare eventuali informazioni in uscita (i.e. una `printf`) o di inserire dati in ingresso.

La parte preponderante della schermata è la zona 3 che è adibita a contenere il codice vero e proprio da modificare ovvero non è altro che l'editor il quale permette all'utente di effettuare le modifiche relative allo sviluppo dell'applicazione di proprio interesse.

In ultimo possiamo individuare la zona 4 contenente la barra dei menù e la barra degli strumenti; la prima è fondamentale in quanto permette di attivare i comandi e le azioni relative al codice nativo ed alle funzionalità dell'ambiente di sviluppo nella loro completezza mentre la seconda non è altro che una barra contenente le scorciatoie alle funzioni più popolari dei menù sopraccitati.

L'insieme di tutti questi componenti e aree di lavoro genera così un ambiente software *user-friendly* dove il programmatore può procedere nello sviluppo dell'applicazione di suo interesse dedicando il minor tempo possibile ad operazioni fuorvianti essendo esse stesse completamente automatizzate e rese disponibili dal *framework* dell'IDE stesso.

3.2. Emulatore

Fig. 3.5 Emulatore JTAG j-link

Mentre l'IDE si occupa sostanzialmente di generare il nostro applicativo, vediamo ora cosa sia necessario al fine di caricare il software sulla memoria interna del nostro microcontrollore e monitorare l'esecuzione dei processi.

L'elemento utile a tale operazione è il JTAG (Joint Test Action Group) che permette così di trasferire le informazioni necessarie tra il PC (o meglio l'ambiente di sviluppo) ed il microcontrollore stesso.

Innanzitutto è bene fornire un'adeguata definizione di cosa sia effettivamente un JTAG; il suo nome proviene da un gruppo industriale denominato Joint Test Action Group nato nel 1985 con lo scopo di sviluppare un metodo efficace per testare i circuiti integrati dopo la produzione. Furono attraversati numerosi livelli e modalità per raggiungere questo scopo sino a che nei primi anni '90 fu sviluppato lo standard IEEE 1149.1 che sarebbe stato chiamato in futuro JTAG prendendo il nome del gruppo che lo aveva appunto progettato.
Per individuare meglio l'utilità dello standard JTAG è bene spendere alcune parole su cosa ci sia alla base di questa tecnologia di test unificata ovvero il *boundary scan*.
Il boundary scan, noto anche come JTAG, fu proposto come soluzione innovativa per far fronte all'aumento vertiginoso della densità dei componenti sulle schede elettroniche e al diffondersi di nuovi tipi di contenitore che rendevano estremamente difficoltoso l'accesso ai punti di misura necessari per eseguire il collaudo quali pins e collegamenti integrati.
Il grande vantaggio della tecnologia boundary scan è dato dal fatto che dedicando un numero molto limitato di piedini, tipicamente cinque (TCK, TMS, TDI, TDO, TRST), e una piccola area di silicio all'interno dei componenti digitali per inserire le logiche di controllo destinate alle funzionalità di collaudo, risulta possibile creare un sistema di verifica adatto a rilevare un gran numero di potenziali guasti sulla scheda, ottenendo un elevato grado di copertura anche quando la maggior parte dei componenti non è raggiungibile per contatto da parte di un tester [16].
Con la necessità di verificare anche il comportamento del codice eseguito dal μC, al boundary scan, si sono aggiunte componenti di controllo nei dispositivi pilotabili attraverso l'interfaccia JTAG; ciò ha reso possibile l'analisi in tempo reale dei valori dei singoli registri del dispositivo (i.e. PC, R1, R2, registri specifici per le periferiche, etc.) e la loro modifica. Tra le ulteriori funzionalità aggiunte, è stremante interessante la possibilità di inserire, all'interno del codice, dei *breakpoint* che permettono, tramite l'interruzione dell'esecuzione di un programma, di osservare in tempo reale che il programma funzioni come previsto; inoltre tramite catene JTAG è possibile "debuggare" il comportamento di più dispositivi presenti sulla scheda.
Stabilito cosa vuol dire JTAG è bene valutare quali siano realmente i mezzi per rendere possibile la comunicazione JTAG tra il computer ed il chip e poter così applicare tutte le attività di diagnostica; ovviamente non si può comunicare JTAG direttamente con un chip tramite un cavo USB ma è necessario uno specifico dispositivo, spesso denominato

anch'esso JTAG o emulatore (Fig. 3.5), che possa trasformare le informazioni provenienti dal computer e dirette al microchip in questione compatibilmente con lo standard JTAG.
Oltre che fornire la possibilità di testare un intero sistema in maniera razionale ed efficace, il JTAG spesso permette di scaricare degli applicativi standard specifici su una data flash permettendo così il caricamento dell'intero programma compilato sulle memorie di un eventuale microcontrollore; questo è proprio quello che fa al caso nostro.
A corredo di ogni JTAG sono presenti ambienti di visualizzazione da installare sul PC; questi possono essere più o meno evoluti e permettono vari livelli di analisi delle informazioni provenienti dal μC.
La scelta del JTAG si basa sostanzialmente sui seguenti fattori:

- Integrabilità con l'IDE: spesso i *drivers* del JTAG installati sul PC permettono l'integrazione con le interfacce grafiche degli ambienti di sviluppo. Questo permette allo sviluppatore di incrociare in maniera intuitiva le informazioni del codice sorgente con quello macchina ed avere così una visione più esplicativa dei registri interni al microcontrollore.
- Funzionalità: un buon JTAG ha la capacita di inserire breakpoint, di effettuare *profiling* ed inoltre avere la capacità di visualizzare più o meno registri relativamente all'architettura del μC.

Nel nostro caso si è scelto il JTAG della IAR j-link che è un piccolo dispositivo hardware ARM JTAG collegabile in maniera *plug&play* direttamente al PC tramite interfaccia USB dal cui altro capo vi è collegato un connettore standard JTAG a 20 pins che troverà la destinazione nella relativa porta sull'Evaluation Board destinata alle attività di collaudo e di caricamento Flash.

Fig. 3.6 Cavo JTAG a 20 pins

In conclusione è bene fare presente che la scelta è stata dettata soprattutto dalla forte integrabilità tra il j-link e l'ambiente di sviluppo IAR Embedded Workbench 5.0.

3.3. Evaluation Board

Come accennato nella parte conclusiva al paragrafo precedente, il cavo JTAG a 20 pins è direttamente collegato al connettore dedicato presente sull'Evaluation Board; vediamo ora meglio nel dettaglio cosa sia una scheda di valutazione e quali siano le sue peculiarità.

Una Evaluation Board è un circuito stampato contenete un microprocessore (nel nostro caso un microcontrollore) e la minima logica di supporto di cui si necessita per testare e valutare le prestazioni del suddetto µC.

Dovendo quindi sviluppare un'applicazione e testarne le relative performance, dovremo dunque disporre di determinate interfacce che rendano possibile il completo utilizzo delle funzionalità offerte dal microchip stesso favorendo così la comunicazione e la gestione degli input e degli output.

Per quanto ci riguarda l'EB fornitaci è la OLIMEX LPC2378-STK in figura della quale seguirà una breve descrizione riguardante le relative interfacce e funzionalità:

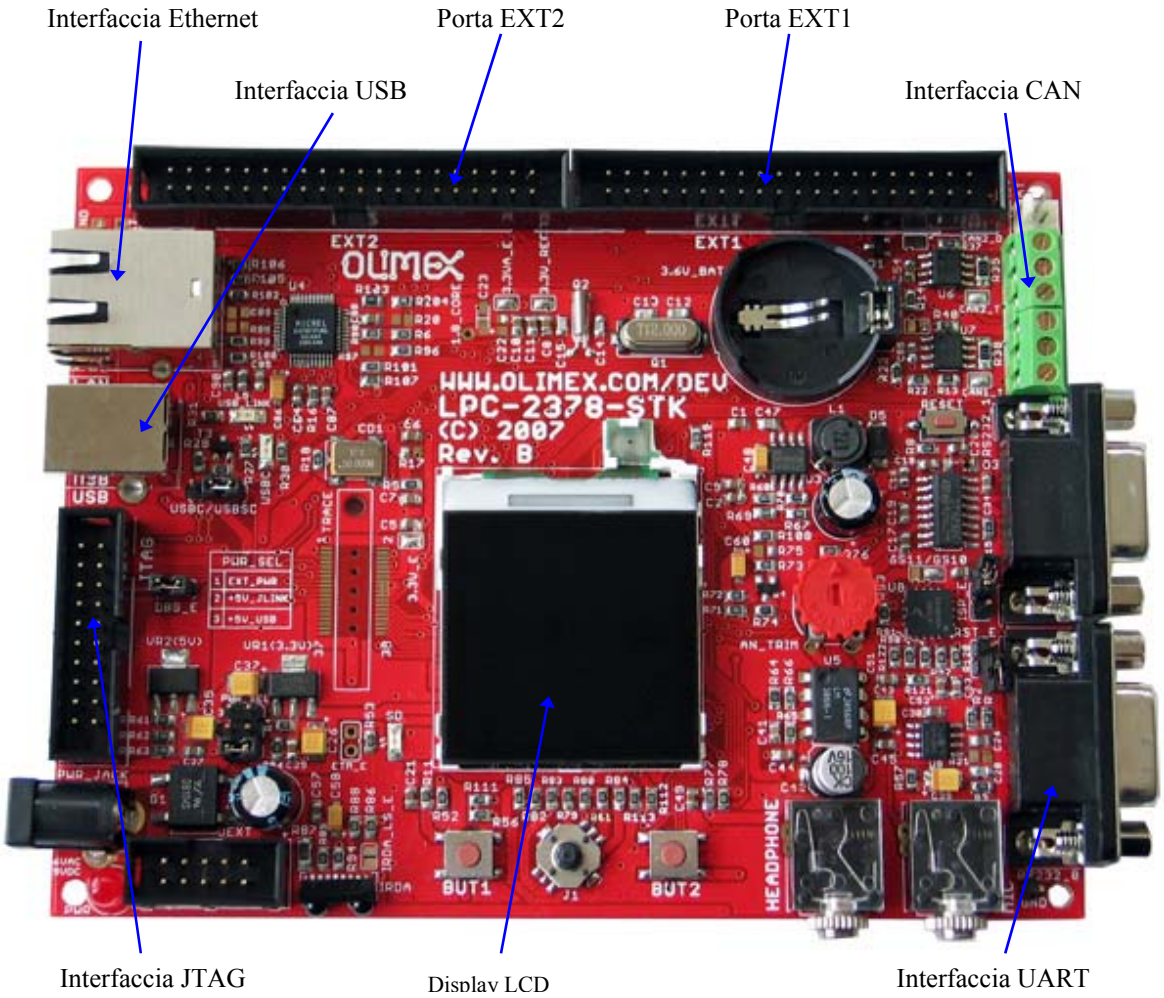


Fig. 3.7 Scheda di valutazione OLIMEX LPC2378-STK

Come visibile in figura questa Evaluation Board mette a disposizione del programmatore una quantità di interfacce notevole ed inoltre diverse connessioni che non sono, però, risultate direttamente utili alle operazioni ed agli scopi di nostro interesse.
Andremo qui di seguito ad elencare tutte le porte dell'EB e successivamente a commentare soltanto ciò la cui partecipazione al progetto è stata di rilevante importanza e, per evitare fastidiosi fraintendimenti, tralasceremo ciò che non è stato di primaria utilità.
Elenchiamo quindi brevemente le caratteristiche della nostra scheda di valutazione OLIMEX avente a bordo (sul retro) il microcontrollore NXP LPC2378 e definiamo gli ulteriori dispositivi *on board* che la caratterizzano:

- Microcontrollore ARM7 NXP LPC2378 (descritto poi al Capitolo 4) posto sul retro
- Alloggiamento per connettore JTAG standard a 20 pins
- Display TFT LCD a colori 128x128 pixel a 12 bit
- Interfaccia Ethernet a 100Mbit/sec
- 2 porte UART seriali RS232
- 2 driver CAN e relativi connettori
- Alloggiamento per SD/MMC Card posto sul retro
- Connettore UEXT contenente le interfacce I^2C, SPI, RS232 ed alimentazione
- Ricetrasmettitore IrDA
- Jacks AudioIn e AudioOut per microfono e cuffie
- Trimmer direttamente connesso all'ADC
- Circuito di RESET con controllo dall'esterno
- 2 regolatori di tensione da 3V e 5V con corrente fino a 800mA
- Alloggiamento per batteria e connettore per RTC
- Dimensioni totali: 135x101mm

Come accennato sopra seguirà ora la descrizione di ciò che di fatto ha contribuito, grazie alla sua presenza sull'EB, al raggiungimento dei nostri scopi sperimentali; in ordine di importanza, la prima interfaccia che incontriamo è quella JTAG che, come già accennato al paragrafo precedente, permette il collegamento con il j-link in maniera tale da poter scambiare tutti i dati necessari ed usufruire di tutte le funzionalità relative ad operazioni di *debugging* e valutazione delle performance.

Dopodichè troviamo immediatamente le porte EXT1 e EXT2 la cui presenza ha permesso, come approfondiremo successivamente al Capitolo 5, l'interscambio di informazioni prettamente elettriche tra il microcontrollore e l'esterno.
Non è così immediato ma, fondamentalmente, l'EB non è altro che un circuito stampato che mette a disposizione ai pins del microcontrollore un'altrettanto numerosa quantità di terminali i quali sono direttamente connessi alle porte sul mondo esterno; per mezzo di tali collegamenti, unitamente alle sopraccitate porte, le periferiche possono dialogare con il microcontrollore stesso e rendere dunque più accessibili le potenzialità di quest'ultimo.
Quindi per rendere un po' più chiaro il discorso possiamo dire che se il µC fosse il serbatoio dell'acqua di un condominio, le porte visibili in figura sarebbero i vari rubinetti nei vari appartamenti e da ognuno di questi l'acqua, ovvero l'informazione, uscirebbe con flussi diversi da rubinetti diversi e con flussi uguali da rubinetti uguali.
Per una maggiore chiarezza nella disposizione elettrica dei vari oggetti sull'EB e per una maggiore predisposizione alla comprensione dei capitoli successivi, forniamo lo schema elettrico (Tavola n°1) dell'intera scheda di valutazione.

3.4. Oscilloscopio

Fig. 3.8 Oscilloscopio LeCroy WaveRunner 6000

In ultima battuta, come visibile in Fig. 3.1, nel nostro percorso di test giungiamo allo oscilloscopio; è venuto dunque il momento di descrivere cosa sia effettivamente questo oggetto e quali siano le sue caratteristiche di maggiore interesse.
L'oscilloscopio è uno strumento elettronico di misura che consente di visualizzare, su di un grafico bidimensionale, l'andamento temporale dei segnali elettrici e di misurare con relativa semplicità tensioni, correnti e potenze. Solitamente l'asse orizzontale del grafico, salvo impostazioni personalizzate, rappresenta il tempo rendendo così l'oscilloscopio adatto ad analizzare grandezze periodiche; per quanto riguarda l'asse verticale, esso individua principalmente valori di tensione.
Ciò che realmente rende possibile la comunicazione tra l'oscilloscopio ed il circuito è costituito da un cavo coassiale ai cui capi vi sono collegati uno speciale connettore

Bayonet Neill Concelman (BNC) atto ad interfacciarsi con l'oscilloscopio, ed uno speciale oggetto denominato sonda da applicare ai capi dei terminali sui quali si ha intenzione di effettuare le misurazioni elettriche legate alle valutazioni ed ai tests di proprio interesse.

Fig. 3.9 Connettore BNC e sonda

Questi strumenti sono inoltre forniti di una consolle di cursori atti ad essere utilizzati per individuare, sullo schermo, istanti di tempo o valori di potenziali con estrema precisione; tali cursori ,inoltre, sono particolarmente utili in quanto permettono di variare la scala delle misurazioni effettuate per renderle così maggiormente interpretabili.

Fig. 3.10 Consolle di cursori

Per quanto ci riguarda, ci è stato fornito un oscilloscopio LeCroy WaveRunner serie 6000 che è stato concepito per essere il miglior oscilloscopio al mondo mai costruito [17].

Esso dispone di un interfaccia utente che rende semplice e rapida la visualizzazione delle misure di proprio interesse; tutti i controlli di visualizzazione e le funzioni base (tra cui il posizionamento dei vari cursori) sono accessibili attraverso le manopole di controllo poste nel layout utente in maniera tale da poter rapidamente zummare e visualizzare i vari dettagli grazie alla presenza di un *touch screen* a colori che coniuga la semplicità con la precisione.

A bordo di questo oscilloscopio gira il sistema Microsoft Windows XP Professional che permette l'utilizzo di un software dedicato alla gestione dell'intero strumento; tale software include una svariata quantità di funzionalità utili ad applicazioni matematiche da apportare al fine di ottenere una ragionevole qualità dei dati ed una conseguente miglior visualizzazione degli stessi; a noi personalmente è stato di estrema utilità, dovendo in seguito occuparci di misurazioni temporali, l'operatore sottrazione sull'asse dei tempi che ci ha permesso di valutare con estrema precisione le tempistiche effettuando una precisa differenza tra i valori di nostro interesse individuati dalle posizioni dei relativi cursori.

Un'altra funzionalità a noi molto utile è stata quella relativa alla possibilità di poter fotografare ed in seguito stampare o salvare su unità rimovibili (vedi report Capitolo 5) ciò che in tempo reale fosse apparso sul display dell'oscilloscopio.

L’oscilloscopio, dunque, si configura come un oggetto prettamente atto all’acquisizione di segnali e, nel nostro caso, anche alla loro parziale elaborazione in maniera tale da rendere possibile una interpretazione temporale dei segnali in termini di livelli di tensione provenienti dal μC da parte dello sviluppatore.

4. Descrizione del Microcontrollore NXP LPC2378

Fig. 4.1 µC NXP LPC23xx

Il mondo dei microcontrollori è estremamente vasto ed insidioso. Questo è dato dal fatto che la quantità dei dispositivi presenti sul mercato sia pressoché innumerevole e ciascuno di questi spesso si differenzi l'uno dall'altro per peculiarità che, solo dopo un'attenta lettura del proprio datasheet, possono essere riconosciute e successivamente interpretate.

Per quanto ci riguarda, la piattaforma utilizzata è il microcontrollore LPC2378 prodotto dalla NXP (founded by Philips) ed appartenente alla famiglia degli ARM7 che individua core di microprocessori *Reduced Instruction Set Computer* (RISC) a 32 bit ottimizzati per applicazioni *power-sensitive* ovvero quegli ambiti in cui il consumo di potenza ha un'importanza rilevante. Essa viene principalmente impiegata per applicazioni di controllo industriale, sistemi elettro-medicali, convertitori di protocollo e sistemi di comunicazione.

Il µC NXP LPC2378 è basato su una CPU ARM7TDMI-S a 16/32 bit con lo scopo di un'emulazione Real-Time combinata ad una memoria Flash ad alta velocità.

Esso è provvisto di un'interfaccia di memoria a 128 bit e, tramite un'architettura comprendente acceleratori hardware, è possibile l'esecuzione di codice a 32 bit alla massima *clock-rate*.

Questo µC è adatto anche alle applicazioni *mission-critical* e per questo è disponibile anche la modalità di utilizzo *thumb* (in questo caso lavoriamo a 16 bit) che permette così di aumentare le prestazioni di circa un 30%.

Inoltre questo dispositivo è ideale per applicazioni a comunicazioni seriali *multi-purpose* in quanto è fornito di numerose tipi di interfacce con l'esterno [18].

4.1. Schema a blocchi

Per poter andare più nel dettaglio riguardo la descrizione del nostro µC è utile fare una breve analisi del suo diagramma a blocchi (Fig. 4.2).

A prima vista può sembrare una rappresentazione piuttosto caotica che però, se adeguatamente compensata da una parte discorsiva che introduca le relative funzionalità più utili ed interessanti, può essere resa di più semplice comprensione.

Andremo dunque ad elencare quali saranno i bus, le dimensioni di memoria, i collegamenti e tutto ciò che possa aiutare ad ottenere una comprensione globale di tale dispositivo per quanto riguarda le caratteristiche e strutturali e comportamentali.

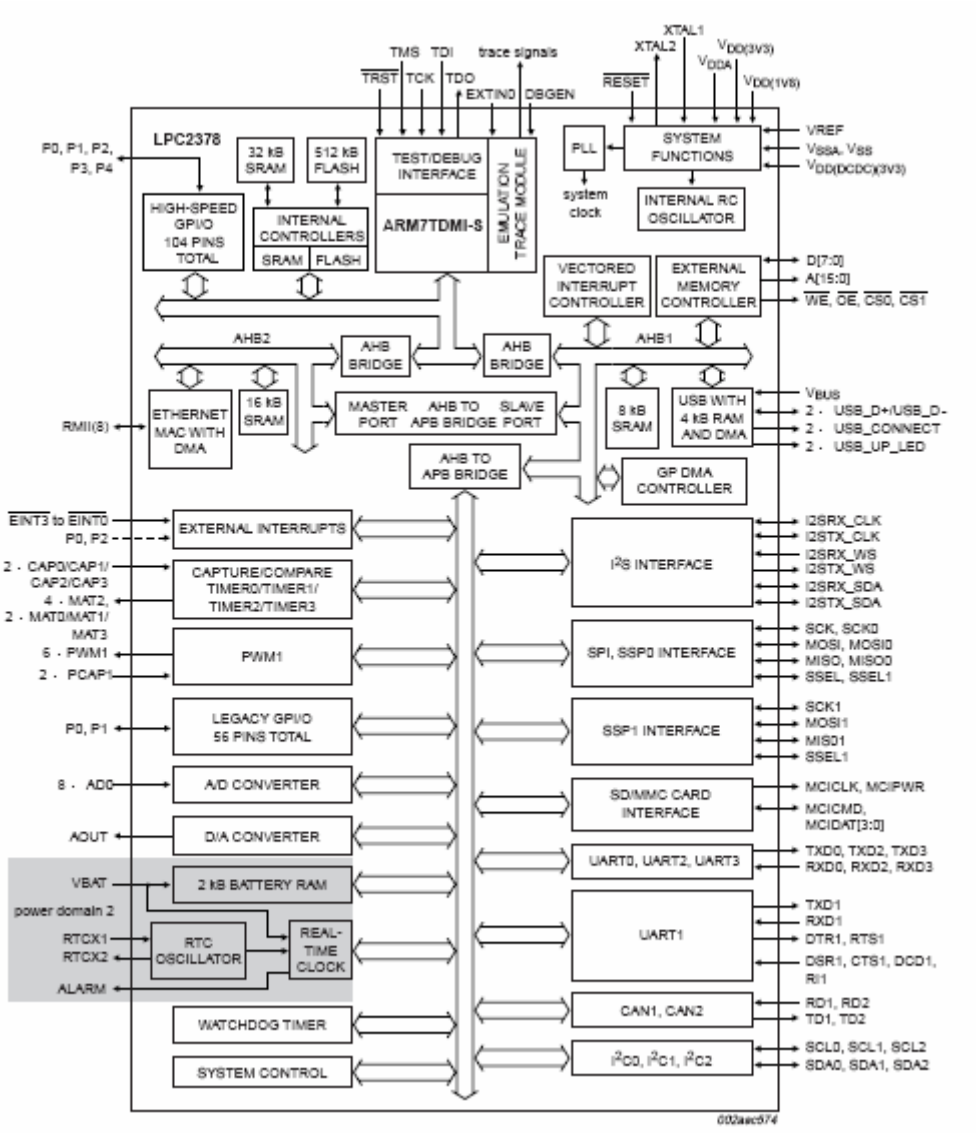


Fig. 4.2 Schema a blocchi del µC LPC2378

4.2. Caratteristiche architetturali del microcontrollore

Come detto al paragrafo precedente, oltre che esplicare quantitativamente le componenti a bordo, è necessario individuare quali siano le peculiarità di tali componenti; analizzeremo quindi le caratteristiche più salienti necessarie per una buona comprensione del dispositivo.

4.2.1. Bus dati AMBA

Oltre alle caratteristiche citate al capitolo 4, il μC è fornito anche di due particolari bus AHB (Advanced High-Performance Bus) e APB (Advanced Peripheral Bus): il primo viene utilizzato per interfacciarsi ad alta velocità con la memoria esterna mentre il secondo si occupa della connessione con le periferiche on-chip.

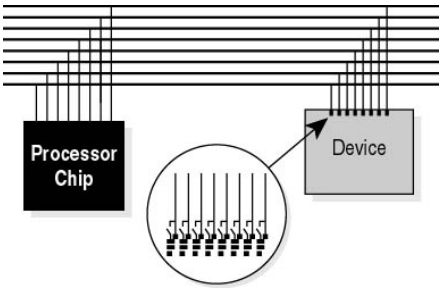


Fig. 4.3 Bus dati generico

Tra le due suddette tipologie di bus sussistono dei *bridge* che hanno lo scopo di agevolare lo scorrimento del traffico ovviando così a percorsi più lunghi e nello steso tempo permettono di evitare un errato flusso di informazioni garantendo una totale indipendenza tra essi; questi sono stati introdotti nel 1996 dall'AMBA (Advanced Microcontroller Bus Architecture).

4.2.2. Memoria Flash programmabile e SRAM statica on-chip

Per quanto riguarda la memoria, il nostro μC è dotato di una memoria Flash incorporata da 512 kB; questa quantità di memoria può essere utilizzata sia per contenere dati che per contenere codice eseguibile.

La memoria può essere programmata in diversi modi attraverso la porta seriale UART (di cui parleremo in seguito). Inoltre l'applicativo può anche cancellare e programmare la Flash mentre l'applicazione è in fase di RUN permettendo così una grande flessibilità sia per quanto riguarda il *data storage*, sia per quanto riguarda il *firmware upgrade*.

Per quanto riguarda la memoria volatile sono stati riservati 32 kB di memoria SRAM esclusivamente per l'ARM core, 16 kB riservati come *buffer* per il controllo dell'Ethernet e 8 kB sono invece dedicati per le operazioni di storage, sia di codice che di dati, del dispositivo USB associato.

In conclusione rimangono 6 kB di memoria SRAM così ripartiti: 4 kB utilizzati in qualità di buffer relativo al controller USB mentre gli altri 2 kB sono relativi all'RTC (Real-Time Clock); questi ultimi hanno la caratteristica di trattenere l'informazione anche quando l'alimentazione è assente, in quanto sono battery-powered.

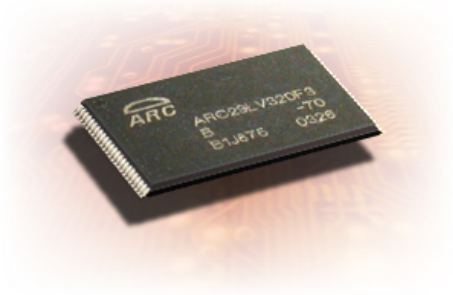


Fig. 4.4 Flash memory generica

Compatibilmente al Bus AMBA AHB viene supportato il GPDMA (General Purpose Direct Memory Access) il quale permette di sfruttare i vantaggi del supporto DMA sul nostro microcontrollore NXP LPC2378.
Il DMA permette ad alcuni sottosistemi hardware di accedere alla memoria di sistema in lettura/scrittura e mette in comunicazione diretta la memoria con le periferiche senza oberare la CPU per tutta la durata dell'operazione ma bensì demandando ad essa solo l'onere di gestire l'inizio e la fine della comunicazione.
In questo contesto il GPDMA abilita transazioni di tipo *peripheral-to-memory*, *memory-to-peripheral*, *peripheral-to-peripheral* e *memory-to-memory*.

4.2.3. Vectored Interrupt Controller (VIC)

Il core del processore ARM può accettare due tipi di interrupt in ingresso chiamati *Interrupt Request* (IRQ) e *Fast Interrupt Request* (FIQ); la differenza sostanziale tra i due è fondata sull'utilizzo e sulla gestione dei registri ovvero, nel modo FIQ, esiste una coppia fisica di alcuni registri utili all'interruzione in maniera tale da ottenere una realizzazione della specifica routine di servizio senza dover salvare tali registri mentre nell'IRQ tale salvataggio risulta doveroso a scapito di una superiore performance [19].

Il μC è in grado di gestire un numero definito di interrupt (nel nostro caso 32 linee) e ad ogni interrupt è associato un pin collegabile ad una sola periferica esterna.
Il microcontrollore verifica lo stato del filo monitorando eventuali fronti di salita o discesa secondo la propria configurazione. Ad ogni interrupt è associato un numero sequenziale utilizzato per indicizzare la VIT (Vectored Interrupt Table). VIT è un vettore di puntatori a funzione che gestiscono gli interrupt associati sulla base della posizione all'interno del vettore stesso. Ad ogni interrupt è associabile una priorità che permette di dipanare eventuali conflitti nella loro gestione; questa, nel nostro caso, è configurabile tra 0 e 15, con 15 come valore di default, che individua la priorità minima.
Dal momento in cui è rilevata una specifica condizione elettrica sul pin di interrupt, il processore, che stava eseguendo operazioni definite dall'applicazione in corso, salva lo stato di tutti i registri e dello stack. In seguito è chiamata la routine della VIT associata all'interrupt che si è verificato; tutto ciò non avviene in modo istantaneo ma in un tempo – come precedentemente detto al paragrafo 2.2.4 – denominato Interrupt Latency.
I pins sulle porte d'input/output programmabili, PORT0 e PORT2 (in tutto 46 pins), possono essere programmati per generare un Interrupt sul fronte di salita, sul fronte di discesa o su entrambi.

4.2.4. Fast General Purpose Parallel I/O

I device pins, non collegati ad una specifica funzione periferica, sono controllati dai registri GPIO (General Purpose Input Output); tali pins possono essere configurati dinamicamente come input o output. Grazie all'utilizzo di registri separati, è possibile il settaggio e la cancellazione di più output simultaneamente. Il GPIO del nostro μC sfrutta alcune caratteristiche di ottimizzazione che ne permettono un'attività accelerata:

- I registri del GPIO sono delocalizzati sul *local bus* dell' ARM in maniera tale da avere un tempo di I/O minore possibile.
- I registri maschera permettono di trattare i settaggi dei bit di porta a gruppi lasciando gli altri bit inalterati.
- Tutti i registri GPIO sono indirizzabili come byte e *half-word*.
- L'intero valore della porta può essere scritto tramite una sola istruzione.

4.2.5. Interfacce on-board

Un'interfaccia di un microprocessore è un dispositivo hardware che si preoccupa in generale della comunicazione tra il microchip stesso e le periferiche o i dispositivi interni.
Le interfacce, infatti, si possono distinguere in:

- *interfacce interne*: queste sono utilizzate per gestire periferiche interne in modo da aumentare le prestazioni del μC (i.e. local bus per le memorie, I^2C per un sensore di temperatura interno, interfaccia dedicata acquisizione dati, etc.).
- *interfacce esterne*: queste, invece, sono utilizzate per gestire la connessione con dispositivi esterni; la connessione è garantita da specifiche di protocollo standard che permettono la comunicazione (i.e. Ethernet supporta lo standard TCP/IP per connettersi con qualsiasi altro dispositivo che implementi lo stesso protocollo).

Il nostro microcontrollore supporta svariate interfacce seriali; in primo luogo, indispensabile per la comunicazione di rete, quella Ethernet, successivamente come collegamento tra diversi dispositivi abbiamo la USB 2.0, strettamente legata al Controller DMA, quella CAN a due canali, quattro di tipo UART, tre I^2C, una I^2S per applicazioni audio digitali, una SPI e due SSP con compatibilità multiprotocollo.
Analizzeremo singolarmente in seguito le peculiarità caratterizzanti di tali interfacce.

4.2.5.1. Interfaccia Ethernet

Ethernet è il tipo più diffuso di rete locale che esista al mondo. Tale sistema consente lo scambio diretto di dati in formato elettronico tra due o più computer senza ricorrere allo scambio di periferiche di archiviazione rimovibili; questo modo di comunicare risulta estremamente vantaggioso qualora i computers siano dislocati in siti molto distanti tra loro.
La natura di Ethernet è quindi di consentire il libero colloquio con qualsiasi macchina collegata e di trasmettere la stessa informazione contemporaneamente a tutte le macchine in ascolto. Chiaramente, come in ogni ambito comunicativo, è necessario stabilire un protocollo di comunicazione e di conseguenza di creare un'adeguata interfaccia compatibile con lo standard Ethernet [20].

Il blocco Ethernet contiene una Ethernet MAC a 10 Mbit/s o 100 Mbit/s completamente caratterizzata da performance ottimizzate attraverso sistemi di accelerazione hardware; questo blocco include una generosa suite di registri di controllo per operazioni *Half o Full-Duplex*, *Flow Control*, *Frame Control*, *Transmission Retry*, *Packet Filtering* e *Wake-Up* su attività LAN.

Fig. 4.5 Interfaccia Ethernet

La CPU ed il blocco Ethernet condividono un sottosistema di accesso dedicato (bus AHB) usato per accedere alla Ethernet SRAM. Tutto il resto del traffico sul bus AHB del microcontrollore avviene su un sottosistema AHB differente dal precedente in maniera tale da separare effettivamente il traffico Ethernet dal restante traffico di sistema.

4.2.5.2. Interfaccia USB

Lo *Universal Serial Bus* (USB) è uno standard di comunicazione seriale che consente di collegare diverse tipologie di periferiche ad un computer; esso è stato progettato per consentire a tipi diversi di periferiche di essere connesse usando un solo tipo di interfaccia standardizzata ed un solo tipo di connettore e per migliorare la funzionalità plug&play consentendo di collegare/scollegare i dispositivi senza riavviare il computer (*hot swap*).
Tecnicamente parlando, si conforma come un bus a 4 fili che supporta la comunicazione tra un host ed un numero di massimo 127 periferiche. L'host controller alloca la banda USB ai dispositivi collegati attraverso un protocollo di gestione basato sul principio del *token*. Un token, in informatica, è un blocco di testo categorizzato che individua in tempo reale quale sia il dispositivo che deve comunicare.

Fig. 4.6 Interfaccia USB

Il controller USB gestisce lo scambio di dati alla velocità di 12 Mbit/s; il motore dell'interfaccia seriale decodifica il data stream USB e successivamente scrive i dati all'appropriato *end-point* ovvero sul buffer della periferica di destinazione; l'avvenuto completamento di un trasferimento USB o l'eventuale condizione di errore, sono indicati attraverso un registro di stato.

4.2.5.3. Interfaccia CAN

Il *Controller Area Network*, noto anche come CAN-bus, è uno standard seriale per bus introdotto negli anni ottanta. Esso è stato espressamente progettato per funzionare senza problemi anche in ambienti fortemente disturbati dalla presenza di onde elettromagnetiche; questo perché il CAN è un bus "di campo" del tutto asincrono ovvero non necessita, come ad esempio l'I^2C, di una linea di sincronizzazione contenente il segnale di clock.

Fig. 4.7 Interfaccia CAN

La linea dati è differenziale bilanciata (conforme allo standard ISO 11898 ovvero operante tra i livelli 1.5V e 3.5V), quindi, un'eventuale disturbo esterno, si ripercuoterebbe in egual misura su entrambi i potenziali. In fase di riconoscimento del livello (alto/basso), viene fatta la differenza dei potenziali dei due poli e conseguentemente il rumore viene,

teoricamente, debellato e tale differenza contiene così l'informazione utile del segnale ovvero il livello che esso assume in quel dato momento [21].
Il blocco contenente il controller CAN è da intendersi in grado di supportare più bus CAN simultaneamente permettendo che il dispositivo sia utilizzato come Gateway, Switch o Router tra un certo numero di bus CAN in applicazioni industriali o automotive.
Nel nostro microcontrollore sono presenti due CAN-bus e due Controllers CAN, un doppio buffer di ricezione e un triplo buffer di trasmissione, inoltre non viene utilizzata la SST (Single Shot Transmission) che non prevede la ritrasmissione dell'informazione.

4.2.5.4. Interfaccia UART

La UART o *Universal Asynchronous Receiver-Transmitter* (ricevitore-trasmettitore asincrono universale) è un dispositivo hardware di uso generale o dedicato. Esso converte flussi di bit di dati da un formato parallelo a un formato seriale asincrono o viceversa.

Fig. 4.8 Interfaccia UART

Oltre alla conversione seriale/parallelo, la UART esegue anche altre operazioni come derivazione, o meglio effetto collaterale, dei suoi compiti primari; il voltaggio usato per rappresentare i bit, viene anch'esso convertito e i bit extra (bit di start e stop) sono aggiunti ad ogni bytes prima che esso venga trasmesso. Inoltre, mentre il rapporto di flusso (byte/s) sul bus parallelo è molto alto, il rapporto di flusso in uscita dalla UART dalla parte della porta seriale è molto più basso. Da notare inoltre come la UART abbia un elenco fisso di rapporti (velocità) che può usare come interfaccia per la sua porta seriale [22].
Praticamente ogni famiglia di microprocessori ha la sua UART dedicata. Nel caso del nostro μC, questa funzione è interna allo stesso; esso, infatti, contiene quattro UARTs e inoltre per rendere standard le linee di dati in trasmissione e ricezione, è stata fornita un'interfaccia capace di controllare completamente l'*handshake*.

4.2.5.5. Interfaccia I^2C

I^2C è acronimo di Inter Integrated Circuit ovvero un sistema di comunicazione seriale bifilare (vedi linee verdi in Fig. 4.9) utilizzato tra circuiti integrati.

Questa interfaccia necessita di un protocollo; in questo caso, esso, prevede una comunicazione seriale con la peculiarità di impegnare solo due linee e permettere, secondo la modalità di funzionamento, velocità di trasmissione elevate (4Mbit/s); analizzando da vicino le due linee possiamo notare che una contiene il segnale di clock e prende il nome di Serial Clock Line (SCL), l'altra, che prende il nome di Serial Data Line (SDA), trasporta l'informazione vera e propria.

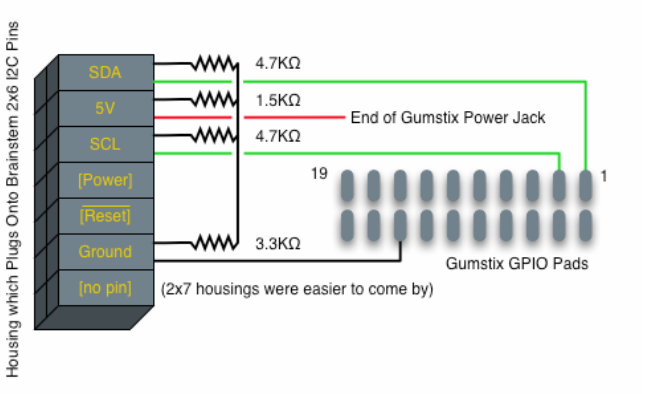


Fig. 4.9 Esempio di schema circuitale interfaccia I^2C

Ogni dispositivo è riconosciuto attraverso un unico indirizzo e può operare o come singolo ricevitore (i.e. display LCD) o come trasmettitore con la capacità sia di ricevere che di trasmettere informazioni (i.e. memoria EPROM).

I ricevitori o i trasmettitori possono operare in modalità *master* o in modalità *slave* dipendentemente da quale sia il chip che inizia la comunicazione.

È da notare, inoltre, come sul nostro µC sia supportata una bit rate fino a 400Kbit/s.

4.2.5.6. Interfaccia I^2S

I^2S è un acronimo che significa *Integrated Interchip Sound* e più specificamente è un'interfaccia standard basata su un bus seriale usato per connettere dispositivi audio digitali tra loro come spesso accade all'interno di un CD player.

Il bus I^2S separa il clock dai segnali contenenti i dati provocando così una connessione caratterizzata da un *jitter* (brusca ed indesiderata variazione di una o più caratteristiche di un segnale) estremamente basso.
A livello architetturale il bus I^2S è provvisto di tre fili collegati a bus seriali che sono il *Data Line*, il *Clock Line* ed il *Word Select Signal*; la connessione base per questa interfaccia ha un master (sempre quello), ed uno slave.
Nel caso del nostro microcontrollore sono forniti canali separati per la trasmissione e la ricezione ed ognuno di essi può operare sia in modalità master che in modalità slave.

4.2.5.7. Interfaccia SPI

Lo SPI o *Serial Peripheral Interface* è un sistema di comunicazione tra un μC ad altri circuiti integrati o direttamente tra più microcontrollori. Esso è un bus standard di comunicazione di tipo seriale, sincrono (per la presenza di un clock che coordina la comunicazione) e Full-Duplex (la comunicazione può avvenire in entrambi i sensi contemporaneamente).
Nell'LPC2378, è contenuto soltanto uno SPI Controller il quale è stato progettato per la gestione di più masters e slaves connessi ad un dato bus; solo un master ed uno slave possono comunicare sull'interfaccia durante il trasferimento del dato. Durante questa attività, nel nostro caso, il master manda sempre da 8 a 16 bit di dati allo slave ed esso manda da 8 a 16 bit di dati al master.

4.2.5.8. Interfaccia SSP

L'SSP o meglio *Synchronous Serial Port*, è un controller in grado di operare su un bus SPI. Nel nostro microcontrollore sono presenti due controller SSP; questi possono interagire con più master e più slave sul bus ma solo un master ed uno slave possono comunicare tra loro durante il trasferimento dei dati [23].
L'SSP supporta comunicazioni e trasferimenti Full-Duplex con frame da 4 a 16 bit che possono fluire dal master allo slave e viceversa, ma in pratica, spesso solo uno di questi flussi trasporta informazioni significative.

4.2.6. Convertitori

I convertitori analogico-digitale (ADC) e i convertitori digitale-analogico (DAC) sono componenti elettronici in grado di mettere in relazione il mondo in cui viviamo, ovvero il mondo analogico, con il mondo in cui "vivono" i calcolatori, ovvero quello digitale.

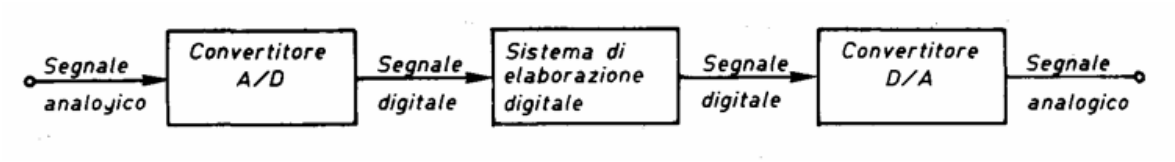


Fig. 4.10 Schema a blocchi per l'elaborazione dei segnali

4.2.6.1. Convertitore ADC

L'ADC, ovvero *Analog (to) Digital Converter*, è un dispositivo elettronico in grado di convertire una grandezza continua (i.e. una tensione) in una serie di valori discreti che individuano, seppur introducendo un errore di quantizzazione, una formulazione numerica della grandezza analogica in ingresso. Fondamentalmente, ci sono due grandezze rilevanti quando si tratta di ADC ovvero la *risoluzione* ed il rapporto *segnale/rumore (S/N)*.
La prima indica il numero di valori discreti che un convertitore può produrre ed è usualmente espressa in bit, la seconda va ad intaccare la prima in quanto se il segnale analogico in ingresso è disturbato da troppo rumore, l'ADC produrrà un valore poco accurato poiché i bit meno significativi saranno funzione del rumore e non del segnale.

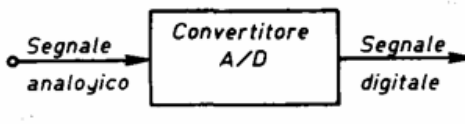


Fig. 4.11 Schema a blocchi di un ADC generico

Inoltre da non sottovalutare è l'importanza che ha la struttura circuitale di conversione; ne esistono diversi tipi tra cui le più importanti sono quelle flash, ad approssimazioni successive, a codifica-delta, a doppia rampa e a *subranging quantiser* [24].

Per quanto riguarda l'LPC2378, la risoluzione è di 10 bit mentre la struttura circuitale di approssimazione è ad approssimazioni successive con 8 canali; utilizzando tale struttura la sua risoluzione è limitata solamente dalle esigenze di *sample-rate* e dal rapporto S/N.

4.2.6.2. Convertitore DAC

Il DAC ovvero *Digital (to) Analog Converter*, è un componente elettronico in grado di produrre una determinata differenza di potenziale in funzione del valore numerico caricato. Le caratteristiche del DAC hanno più o meno rilevanza a seconda dell'impiego; le più importanti di queste peculiarità sono la *risoluzione* e la *velocità di elaborazione*.
La prima è estremamente importante per le misure di precisione (i.e. produzione di brani musicali ad alta fedeltà) mentre la seconda è strettamente legata alla prima in quanto deve essere sufficientemente alta per far si che la risoluzione scelta possa essere supportata.

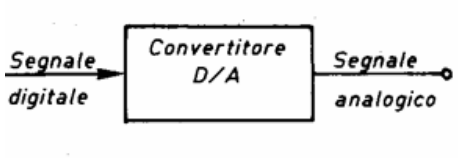


Fig. 4.12 Schema a blocchi di un DAC generico

Come nel convertitore precedente, anche in questo caso ci sono diverse implementazioni circuitali per raggiungere lo scopo; le più diffuse sono quelle a rete R-2R, a CMOS moltiplicante ed a capacità commutate. È necessario, inoltre, introdurre il valore di *fondo scala* che rappresenta il limite superiore del segnale prodotto ed effettivamente misurabile da un determinato strumento di misura.
Per quanto riguarda il nostro μC la risoluzione è mantenuta costante a 10 bit mentre il valore di fondo scala è definibile dall'utente in base all'applicazione [25].

4.2.7. Oscillatori

Un oscillatore è un circuito elettronico che genera forme d'onda di frequenza, forma ed ampiezza di molteplici tipi senza un segnale d'ingresso. Gli oscillatori, nella loro vastità, sono impiegati in innumerevoli applicazioni che spaziano dalla temporizzazione di circuiti digitali e non alla generazione di portanti per le telecomunicazioni.

Il nostro μC contiene tre oscillatori indipendenti a cristallo: il *Main Oscillator*, l'*Internal RC Oscillator* e l'*RTC Oscillator*.
Ognuno di questi può essere usato per più di uno scopo e ciascuna di queste sorgenti di clock può essere scelta via software per pilotare in primo luogo il PLL e successivamente la CPU.

Fig. 4.13 Oscillatore al quarzo

È bene notare dunque come, negli apparati elettronici dove sono presenti oscillatori, spesso venga comunemente utilizzato un Phase Locked Loop (PLL) ovvero un anello ad aggancio di fase; questo non è altro che un circuito elettronico progettato per generare un'onda di una specifica frequenza, sincronizzata con un'onda di valore differente fornita in ingresso.
Il PLL dell'LPC2378 accetta segnali di clock in ingresso nel range di frequenze tra i 32KHz e i 50MHz; la frequenza in ingresso è moltiplicata per un certo valore in maniera tale da ottenere un'alta frequenza e successivamente ridivisa per fornire il clock usato dalla CPU e dal blocco USB.

4.2.7.1. Main Oscillator

Il *Main Oscillator* o oscillatore principale, viene utilizzato come sorgente di clock per la CPU, con o senza l'ausilio del PLL.
Questo oscillatore a cristallo opera nel range di frequenze da 1MHZ a 24MHz; questa frequenza può essere incrementata sino al massimo valore operativo della CPU dal PLL.
In questo modo è possibile, utilizzando un oscillatore generico, riuscire ad ottenere la frequenza desiderata per le proprie necessità operative agendo di fatto solo sul PLL.

4.2.7.2. Internal RC Oscillator (IRC)

L'*Internal RC Oscillator* viene utilizzato come sorgente di clock per quanto riguarda il *Watchdog Timer* e/o come sorgente di clock pilotante il PLL e successivamente la CPU; la frequenza nominale dell'IRC è 4MHz con l'1% di accuratezza.
Nella fase di accensione o in qualsiasi fase di reset del chip, l'LPC2378 utilizza l'IRC come sorgente di clock e, successivamente, si può commutare la sorgente verso una fonte di oscillazione differente.

4.2.7.3. RTC Oscillator

L'*RTC Oscillator* o *Real-Time Clock Oscillator* è un oscillatore atto a generare un segnale di clock per il dispositivo con funzione di orologio del nostro microprocessore ovvero fornisce la temporizzazione al dispositivo che conteggia il tempo reale (anno, mese, giorno, ora, minuto,secondo) anche successivamente alla fase di spegnimento del sistema. Per poterlo fare, gli RTC hanno un oscillatore a quarzo, a loro dedicato, e sono alimentati da una speciale batteria autonoma rispetto all'alimentazione generale.
Al contrario, i clock non Real-Time, non funzionano quando il sistema è spento. Anche questo oscillatore può essere utilizzato come sorgente di segnale per il Watchdog Timer ed in casi poco frequenti anche come fonte del clock della CPU, chiaramente dopo un adeguato passaggio attraverso il PLL.

4.2.8. Counters/Timers

I Counters sono componenti elettronici costituiti da un chip in cui sono implementate le funzioni di contatore digitale; spesso è possibile realizzare questa funzione anche impiegando soltanto semplici dispositivi "di registro" quali possono essere i *flip-flop*.
Tra i vari contatori ci sono anche i timer digitali che sono orologi con funzioni specializzate nel misurare lo scorrere del tempo con una grande precisione; nei circuiti integrati sono fatti di logica digitale e sono implementati come sistemi a *single-chip*.
Nel nostro microcontrollore sono presenti quattro dispositivi di questo tipo ovvero il *General Purpose 32-bit Timer*, il *Pulse Width Modulator* (PWM), il Watchdog Timer ed il *Wake-Up Timer*.
Analizzeremo di seguito le caratteristiche principali che individuano i suddetti dispositivi.

4.2.8.1. General Purpose 32-bit Timer/external event Counter

Nei microcontrollori, i timers caratterizzano la maggior parte delle operazioni di controllo misurando più specificamente, sulla base dei cicli di clock, il tempo trascorso. Quanto appena detto vale anche per i counters con l'unica differenza che, piuttosto che calcolare il tempo trascorso, sono dedicati al conteggio di eventi esterni; c'è da dire , però, che i nomi Timer e Counter possono essere utilizzati in maniera intercambiabile quando parliamo di componenti hardware [26].

Nell'LPC2378, sono inclusi quattro 32-bit Timer/Counters; questi sono progettati per contare i cicli del sistema che derivano dall'utilizzo di un clock generato esternamente in qualità di ingresso. Questo genere di dispositivi può anche, in via opzionale, generare interrupt o eseguire altre azioni in determinati valori assunti dal timer.

I Timer/Counter, inoltre, sono provvisti di quattro "ingressi di cattura" i quali interdicono il valore del Timer quando vi è una transizione del segnale di input ed, eventualmente, generare un interrupt.

4.2.8.2. Pulse Width Modulator (PWM)

Generalmente per PWM si intende quel dispositivo che effettua la modulazione di larghezza di impulso (Pulse Width Modulation); questo tipo di modulazione è digitale e più specificamente l'informazione è codificata sotto forma di durata nel tempo di ciascun impulso di un segnale.

La PWM è basata sul blocco Timer standard e ne eredita tutte le sue funzioni ed inoltre in addizione a quelle ne aggiunge altre basate sul confronto tra gli eventi di registro; il Timer è progettato per contare i cicli derivanti dal clock di sistema.

Il nocciolo della questione è che la PWM può essere utilizzata in uno svariato numero di applicazioni come ad esempio nel controllo di motori elettrici multifase grazie alla sua peculiarità di poter controllare separatamente la locazione dei fronti di salita e di discesa; per fare ciò, si sfruttano adeguatamente dei registri che caratterizzano la maniera in cui debba essere impostata la PWM.

Nel nostro μC vi è un solo blocco PWM con impiego su operazioni Counter o Timer che può utilizzare il clock periferico o uno degli ingressi di cattura come sorgente di clock ed inoltre può essere impiegato come Timer standard se la modalità PWM fosse disabilitata.

4.2.8.3. Watchdog Timer (WDT)

Il Watchdog Timer, ovvero il temporizzatore di supervisione, è un sistema di temporizzazione hardware che permette alla CPU la rilevazione di un loop infinito di programma o di una situazione di *deadlock* (due o più task si bloccano a vicenda aspettando che uno esegua una certa azione che serve all'altro e viceversa).
Tale rilevazione può consentire di prendere dei provvedimenti per correggere la situazione, generalmente effettuando un reset del sistema, evitando così di finire in uno stato erroneo.
Quando il WDT è abilitato, esso genererà un *system reset* se l'applicativo utilizzatore non ricarica (resetta) il Watchdog prima di un determinato ammantare di tempo.
Nell'LPC2378 il WDT può essere utilizzato in modalità debug ed inoltre questo μC è fornito di un *flag* che indica la presenza o meno di un'operazione di reset del WDC stesso.

4.2.8.4. Wake-Up Timer

La funzione fondamentale del Wake-Up Timer è quella di monitorare il cristallo oscillatore con lo scopo di verificare, qualora possa essere avviata, l'esecuzione del codice; quando viene fornita potenza al chip o quando qualche evento causa l'uscita del chip dalla sospensione, l'oscillatore necessita di una breve quantità di tempo per produrre un segnale di ampiezza sufficiente a pilotare lo logica di clock. L'ammontare di questo tempo dipende da numerosi fattori quali possono essere il transitorio rampa della V_{DD} (tensione di alimentazione), il tipo di cristallo ed eventuali capacità parassite provocate da eventuali dispersioni dei parametri nei componenti.
Il nostro μC utilizza un IRC oscillator stabilizzato a 4MHz in qualità di sorgente di clock; questo permette al chip, in stato di sospensione, di riattivarsi rapidamente lasciando così, all'oscillatore principale ed al PLL, il tempo necessario per avviarsi e stabilizzarsi.

5. Verifica dei parametri di confronto dei due RTOS

Come accennato nella parte conclusiva del Capitolo 2, andremo ora ad analizzare approfonditamente i due RTOS prescelti per decretare, dulcis in fundo, quale sia il sistema operativo più adatto alle nostre esigenze.
Più specificatamente, i parametri di nostro interesse sono il Context Switch Time, l'Interrupt Latency ed il Footprint; i primi due necessiteranno dunque di un'analisi temporale particolarmente accurata mentre il terzo ed ultimo parametro sarà descritto tramite un'analisi dimensionale rappresentando esso stesso un'occupazione di memoria.
Per fare ciò, ci siamo serviti di tutti i componenti elencati al Capitolo 3; in questo modo abbiamo così a disposizione i due RTOS da mettere a confronto e gli strumenti sia software che hardware necessari ad eseguire le dovute valutazioni sperimentali.
Per fare queste verifiche ci siamo resi conto che era necessario lo svolgimento di una prassi costituita da tre punti cardine che vanno dal criterio sperimentale al report dei risultati ottenuti, passando attraverso una più ampia fase nella quale si è svolta l'implementazione di un algoritmo atto all'acquisizione delle misurazioni di interesse.

- **Criterio sperimentale**: per criterio sperimentale intendiamo una descrizione introduttiva del percorso generale che abbiamo intrapreso dettagliando, ove utile, i vari noccioli caratterizzanti questa sessione di calcolo.
- **Algoritmo implementato**: per algoritmo implementato si intendono tutte quelle modifiche e tutte quelle aggiunte al codice nativo del sistema operativo che permettono di svolgere attività di test e sperimentazione. Ci siamo accorti che, nell'effettuare le varie operazioni, è bene procedere per piccoli moduli in maniera tale da fissare, volta dopo volta, successivi obiettivi che, sommati, portino al raggiungimento del traguardo preposto.
- **Report dei risultati**: nel report dei risultati andremo a definire in maniera puramente informativa gli effettivi valori ottenuti in seguito alle suddette operazioni di valutazione; questo è stato reso possibile grazie anche alle notevoli capacità funzionali degli strumenti a noi messi a disposizione da ELSAG Datamat.

Mettiamoci all'opera.

5.1. Context Switch Time

Riprendendo brevemente quanto detto al paragrafo 2.2.3 relativamente alla descrizione del CST, dobbiamo ora verificare sperimentalmente quanto sia il tempo di commutazione di contesto che impiegano effettivamente i due RTOS messi a confronto.

5.1.1. Criterio sperimentale

Come già accennato al paragrafo 3.4, per poter calcolare, con la dovuta precisione, un valore tanto rapido (nell'ordine dei μs), abbiamo scelto l'oscilloscopio in quanto strumento in grado di individuare variazioni elettriche ed i relativi tempi di mutamento del segnale con la dovuta precisione.
A questo punto sorge spontanea una domanda: come legare l'oscilloscopio al calcolo di un valore temporale che nasce e muore nel mondo software?
La problematica è stata affrontata analizzando quali meccanismi del μC potessero evidenziare un evento software come un segnale elettrico; la soluzione prevede che il software gestisca lo stato di un segnale d'uscita attivandolo in determinati istanti di nostro interesse e, fatto questo, non resta che scegliere quale output debba essere pilotato dal sistema in maniera tale da avere un riscontro su ciò che accade all'interno del μC.
Dando così, in uscita al microcontrollore, e di conseguenza ad una porta d'uscita dell'EB, un fronte di salita/discesa generato da una porzione di codice opportuna, atta ad individuare un cambio di contesto, possiamo delineare sull'asse dei tempi dello oscilloscopio l'effettivo costo del CST in termini di ritardo operativo.
Un metodo "smart" per fare ciò è quello di far accendere/spegnere un led, connesso ad un pin di una porta dell'Evaluation Board, ogniqualvolta avvenga un cambio di contesto all'interno del sistema operativo.

5.1.2. Algoritmo implementato

In questa fase di modifica del codice sorgente del sistema operativo, il primo obiettivo è stato quello di riuscire a pilotare un GPIO che si comportasse come uscita su una porta a noi favorevole al posizionamento dei dispositivi per la raccolta delle informazioni puramente elettriche.

Ricercando sullo User Manual [27] abbiamo notato che sul μC ci sono alcuni pins configurabili dinamicamente come input o output; per fare in modo che questi pins siano effettivamente configurati in qualità di GPIO, nel μC sono stati disposti dei registri a 32 bit che, se settati adeguatamente, permettono di abilitare i relativi pins in modalità GPIO.
Analizzando lo schema elettrico dell'EB (Tavola n° 1) abbiamo rilevato sul μC la presenza del pin 0 della porta 4 (pin P4.0) che può essere configurato come I/O; tale pin inoltre è facilmente raggiungile con una sonda, applicata ad un connettore specifico sulla EB, in quanto quest'ultimo è direttamente connesso al suddetto pin tramite una circuiteria dedicata.
Questo pin è configurabile, per quanto riguarda la sua funzione, mediante il registro **PINSEL8** rappresentato schematicamente di seguito:

REGISTRO: PINSEL8															
31:30	29:28	27:26	25:24	23:22	21:20	19:18	17:16	15:14	13:12	11:10	9:8	7:6	5:4	3:2	**1:0**
P4.15	P4.14	P4.13	P4.12	P4.11	P4.10	P4.9	P4.8	P4.7	P4.6	P4.5	P4.4	P4.3	P4.2	P4.1	**P4.0**

Tab. 5.1 Astrazione schematica del registro PINSEL8

Settando i due bit meno significativi (evidenziati in tabella) entrambi a 0, si pone il pin 0 della porta 4 con funzionalità di GPIO; in maniera del tutto analoga, settando differentemente le 16 coppie di bit presenti nel suddetto registro, esso può far assumere, ai pins corrispondenti, funzionalità diverse.
Passiamo ora all'analisi del registro atto a configurare il pin P4.0 relativamente alla possibilità che esso assuma dati in ingresso o fornisca dati in uscita; tale registro prende il nome di **FIO4DIRL** ed è strutturato come mostrato nella tabella sottostante:

REGISTRO: FIO4DIRL									
31	30	29	……	16	15	……	2	1	**0**
P4.31	P4.30	P4.29	……	P4.16	P4.15	……	P4.2	P4.1	**P4.0**

Tab. 5.2 Astrazione schematica del registro FIO4DIRL

Il bit del registro sopra che permette di decidere la direzione dei dati passanti attraverso il pin P4.0 è quello meno significativo (LSB); per configurare la direzione come ingresso si pone tale bit a 0, viceversa per configurare la direzione in uscita si pone a 1.

Per quanto ci riguarda, dovendo pilotare un LED, dobbiamo inviare in uscita un segnale atto a procurare la sua accensione; quindi porremo l'LSB a 1.
Un altro registro utile è il **FIO4PIN** che setta lo stato del pin P4.0; come nei casi precedenti inseriamo una tabella esplicativa relativa alla struttura di tale registro:

REGISTRO: FIO4PIN									
31	30	29	.....	16	15	.....	2	1	**0**
P4.31	P4.30	P4.29	.....	P4.16	P4.15	.....	P4.2	P4.1	**P4.0**

Tab. 5.3 Astrazione schematica del registro FIO4PIN

Questo registro configura lo stato dei pins della porta 4 del microcontrollore ad un livello di tensione positivo, ponendo i bit a 1, o a massa ponendoli a 0; nel nostro caso settiamo il primo bit in maniera concorde allo stato voluto.
Da notare inoltre come si debba prestare attenzione a non alterare lo stato dei bit non interessati all'azione attuale quando si vuole modificare i restanti bit dei registri.
Nello schema elettrico della scheda embedded OLIMEX LPC2378-STK (Tavola n° 1), abbiamo visto che il pin P4.0 del µC corrisponde in maniera biunivoca al pin 24 della porta EXT1. Inoltre, sempre sulla EXT1, sono disponibili alcuni pins notevoli che sono collegati ad altrettanti valori di tensione notevoli; tra questi si evidenzia, per nostro interesse, il pin EXT1-40 connesso a massa.

Fig. 5.1 Zoom su porta EXT1 dell'EB

Da qui si può concludere, per quanto riguarda il pilotaggio del GPIO, che sono stati acquisiti elementi sufficienti per poter collegare un led ai due pins sopra citati della porta EXT1 in modo tale da rilevare il cambio di contesto e portarci, così, al raggiungimento di una parte del primo obiettivo. Il dispositivo contenente il led è così costituito:

- Un LED verde
- Una resistenza di protezione da 10Ω
- Uno zoccolo nero compatibile con la porta EXT1

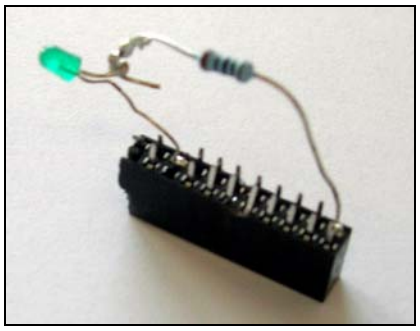


Fig. 5.2 Dispositivo LED

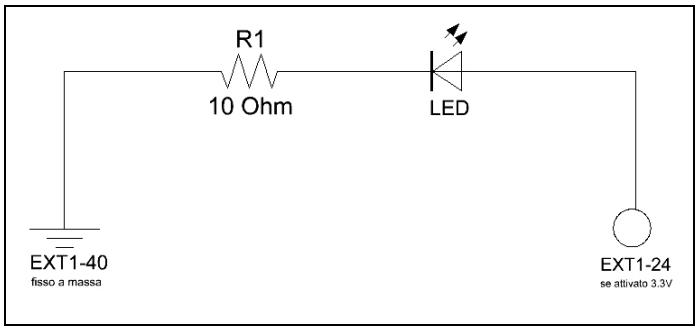


Fig. 5.3 Schema circuitale del dispositivo LED

Il led e la resistenza sono connessi in serie e i loro terminali sono collegati direttamente sullo zoccolo in corrispondenza degli alloggiamenti previsti per i pin EXT1-24 e EXT1-40.
Per quanto riguarda il codice, abbiamo creato un'applicazione contenete due tasks eseguiti in maniera alternata da parte dello scheduler; abbiamo in seguito introdotto una funzione che inizializzasse il GPIO di nostro interesse, successivamente abbiamo individuato la routine identificante l'effettivo cambio di contesto ed infine abbiamo inserito, al suo interno, alcune istruzioni per alzare/abbassare il segnale sul led in corrispondenza della funzione atta al cambio di contesto.
Tutto ciò è stato effettuato e per FreeRTOS e per IAR PowerPac prestando comunque attenzione ad individuare le rispettive porzioni di codice con la dovuta coerenza funzionale, così da ridurre al minimo le discrepanze non relative all'effettive performance e poter così rendere confrontabili e due risultati.
Riporteremo di seguito le porzioni più significative del codice utilizzato mentre ci limiteremo a citare i prototipi e le righe realmente utili delle funzioni qualora siano particolarmente lunghe o di importanza non necessaria alle operazioni prettamente di analisi.

5.1.2.1. FreeRTOS

Il primo sistema operativo analizzato è FreeRTOS; esso è reperibile all'indirizzo internet http://www.freertos.org ed inoltre sono proposti in rete numerosi porting sia relativi a diverse piattaforme hardware (nel nostro caso ARM7), sia relativi a differenti ambienti di sviluppo (nel nostro caso non c'era quello di nostro interesse).

Come primo passo abbiamo effettuato il porting di tale sistema nell'ambiente di sviluppo IAR Embedded Workbench 5.0 per LPC2378 partendo da uno reperito in rete ed interamente progettato per un *workspace* di Eclipse. Una volta effettuato tale porting e verificata la completa e corretta compilazione dell'intero RTOS, da parte del compilatore in ambiente IAR, siamo passati direttamente alla fase di modifica/aggiunta del codice utile ai nostri interessi sperimentali.
Come precedentemente anticipato abbiamo dovuto cercare in rete una funzione che ci indicasse quale fosse la modalità di inizializzazione dei GPIO; la firma di questa funzione è `void GPIOInit( DWORD PortNum, DWORD PortType, DWORD PortDir, DWORD Mask)` ed i suoi parametri, come si può vedere nella dichiarazione del prototipo, sono `PortNum` che indica il numero di porta, `PortType` che si riferisce all'eventuale utilizzo di porte fast o porte regolari, `PortDir` che indica se la porta viene considerata d'ingresso o d'uscita e `Mask` che è il valore assunto dalla maschera utilizzata per effettuare successivamente uno XOR.
Il codice proveniente da terze parti, spesso, presenta alcune problematiche:

- *Porting*: il codice può essere pensato per architetture differenti e quindi le primitive possono risultare diverse da quelle esportate dal nostro sistema operativo.
- *Configuration*: il codice può essere pensato per molte architetture e quindi bisogna configurarlo relativamente alla piattaforma in utilizzo.

Questo genera delle incongruenze nella corrispondenza dei tipi primitivi e non, provocando così numerosi errori di prima compilazione; nella maggior parte dei casi questi errori possono essere ovviati per mezzo di un surplus di codice che vada a sciogliere i dubbi sulla porzione importata sconosciuta.
Solitamente in questi casi questo problema si gestisce aggiungendo nell'intestazione del file apposite `#define` che possano mettere in corrispondenza l'esattezza procedurale del codice importato con i valori esatti che devono assumere tali variabili/costanti nel contesto del RTOS in cui vengono ospitate. Chiaramente non si tratta solo di semplici `#define` ma ciò che deve essere aggiunto può appartenere a diverse nature quali funzioni, files header ed altre strutture non ora specificate.
Vediamo ora l'*entry point* del nostro programma ovvero il `main.c` discutendone le principali caratteristiche di funzionamento:

```
[...]

#define GPIO_SC  (1UL<<0)
#define GPIO_SC_FIO  FIO4PIN

[...]

//ENTRY POINT
int main( void )

{
     //Init Board

     printf("INIT BOARD...\n");
     prvSetupHardware();
     BSP_Init();

     //Init GPIO

     GPIOInit( 4, FAST_PORT, DIR_OUT, GPIO_SC);

     //Create the task, storing the handle.

     printf("START TASK...\n");
     xTaskCreate( Task_1, "TASK1", mainBASIC_WEB_STACK_SIZE, NULL,2,&xHandle1 );
     xTaskCreate( Task_2, "TASK2", mainBASIC_WEB_STACK_SIZE, NULL,2,&xHandle2 );

     //Start the scheduler

     printf("START SCHEDULER...\n");
     vTaskStartScheduler();

     //Non deve arrivare qui!!!

     printf("END!!\n");

     return 0;
}
```

Si può osservare come la suddetta funzione sia composta da cinque blocchi contraddistinti da altrettanti commenti che ne individuano la natura; il primo di questi è `//Init Board` contenente tre istruzioni utili all'inizializzazione della scheda di valutazione, il secondo `//Init GPIO` contiene una sola istruzione ed è la chiamata alla funzione di inizializzazione del GPIO già esposta. Il terzo `//Create the task, storing the handle` contiene tre istruzioni di cui, la prima, è un banale `printf` atto a rilevare il passaggio in questa zona di codice in fase di esecuzione, mentre le altre due sono le chiamate alla funzione di creazione dei due tasks. Il quarto `//Start the scheduler` contiene due istruzioni di cui, quella realmente utile, è la chiamata alla funzione `vTaskStartScheduler()` che avvia la procedura di schedulazione dei tasks e, una volta entratovi, non dovrebbe più uscirne; da qui il quinto ed ultimo blocco `//Non deve arrivare qui!!!` che è da considerarsi di sicurezza in quanto è atto a rilevare un'eventuale esecuzione di codice errata.

Entriamo ora nel nocciolo della questione senza perdere di vista il vero scopo della nostra analisi ovvero il calcolo del CST; il problema fondamentale che abbiamo riscontrato è stato quello di riuscire ad individuare quale fosse effettivamente la funzione praticante questa operazione. Dopo numerose ricerche in rete e discussioni con gli ingegneri Matteo Cantarini e Luigi Veardo, abbiamo visto che la funzione relativa al cambio di contesto risulta essere la seguente:

```
void vPortPreemptiveTick( void )
      {
                GPIO_SC_FIO ^= GPIO_SC; //semino-luciani
                GPIO_SC_FIO ^= GPIO_SC; //semino-luciani

             /* Increment the tick counter. */
             vTaskIncrementTick();

      /* The new tick value might unblock a task.  Ensure the highest task that
      is ready to execute is the task that will execute when the tick ISR
      exits. */

             vTaskSwitchContext();

             /* Ready for the next interrupt. */
             T0IR = portTIMER_MATCH_ISR_BIT;
             VICVectAddr = portCLEAR_VIC_INTERRUPT;

                GPIO_SC_FIO ^= GPIO_SC; //semino-luciani
                GPIO_SC_FIO ^= GPIO_SC; //semino-luciani
      }
```

In questa porzione di codice possiamo osservare che è contenuta la chiamata alla funzione `vTaskSwitchContext()` la quale è la funzione a più basso livello relativa al cambio di contesto ma non sufficiente ad attuare l'operazione completa; a questo punto il più è fatto.
Inseriamo, dunque, ulteriori righe di codice con lo scopo di pilotare il GPIO che ci eravamo preposti ovvero il pin 24 sulla porta EXT1 dell' Evaluation Board. Vale la pena analizzare cosa significhi la riga di codice `GPIO_SC_FIO ^= GPIO_SC` che vediamo ripetuta quattro volte all'interno del listato.
L'operatore `^` implementa l'operazione di XOR bit a bit tra due variabili, in questo caso registri, di uguale dimensione; poiché non intendevamo apportare modifiche ai bit non direttamente coinvolti alle operazioni di nostro interesse, abbiamo dovuto aggirare in qualche maniera il problema.

XOR		
X	Y	U
0	0	0
0	1	1
1	0	1
1	1	0

Tab. 5.4 Tabella di verità dello XOR

Considerando il segnale X come controllo si può rilevare la seguente proprietà:

$$(X = 0) \rightarrow U = Y$$

$$(X = 1) \rightarrow U = \overline{Y}$$

Sulla base di questa proprietà se applichiamo come controllo X pari a 0 e come ingresso Y, in uscita otterremo il valore di Y stesso; altresì se applichiamo come controllo X pari a 1 e diamo come ingresso Y, come uscita otterremo il valore di Y negato.

A questo punto, poiché ci interessa modificare lo stato del P4.0, sarà necessario utilizzare una maschera per lo XOR con valore pari a 0x1; questo lascerà inalterati tutti i bit tranne il primo ossia quello relativo al pin 0.

Quindi rientrando nel contesto dell'analisi del codice, l'istruzione `GPIO_SC_FIO` individua una `define` relativa al registro `FIO4PIN` mentre l'istruzione `GPIO_SC` individua una `define` della maschera; tra i due verrà dunque effettuato lo XOR ed ogniqualvolta questa istruzione compaia nel codice, avverrà una transizione di livello direttamente dipendente dallo stato precedete.

In questo modo riusciamo, per ogni coppia di queste istruzioni, ad ottenere un'accensione/spegnimento del led estremamente rapidi (basati appunto sull'operazione di XOR che genera fronti di salita/discesa) a tal punto che l'occhio umano non sia in grado di rendersene conto ma, per quanto riguarda il segnale elettrico rilevato dall'oscilloscopio, questa coppia di istruzioni risulta estremamente utile in quanto genera un picco di segnale tendente alla tensione di alimentazione del μC.

È bene notare, inoltre, come questo sali/scendi del segnale possa essere effettivamente considerato un picco poiché lo *skew* (ritardo di commutazione originato sul segnale dovuto alla tipologia del sistema) del nostro dispositivo elettronico sia estremamente ridotto. Quanto affermato per le due suddette istruzioni, vale anche per la successiva coppia.

Dopo aver effettuato le modifiche esposte in precedenza, abbiamo avviato una ricompilazione totale del progetto e successivamente, attraverso gli strumenti di debug, abbiamo portato il tutto sulla memoria Flash del microcontrollore.
Da qui, finalmente, si può asserire che il CST sia proprio l'intervallo temporale che intercorre tra i due picchi di segnale rilevati posizionando la sonda dell'oscilloscopio ai capi del led.

5.1.2.2. IAR PowerPac

L'altro sistema operativo da noi analizzato è IAR PowerPac; esso, avendo licenza proprietaria e quindi non liberamente scaricabile dalla rete, ci è stato fornito direttamente da ELSAG Datamat assieme a tutto il supporto riguardante IAR compresi manuali, JTAG, EB ed ambiente di sviluppo. C'è da notare inoltre che, per quanto riguarda questo RTOS, non è stato necessario alcun porting su piattaforma IAR in quanto esso era già disponibile.
Come nel caso precedente, anche qui ci siamo mossi per piccoli passi e per lo più abbiamo svolto le stesse operazioni necessarie allo sviluppo del test con il sistema precedente; alla stessa maniera si è reso indispensabile inizialmente pilotare il medesimo GPIO collegato al led di valutazione e conseguentemente a questo è stato necessario risolvere, per quanto possibile, tutti quei conflitti e quelle incoerenze derivanti dall'importazione della funzione relativa al pilotaggio del GPIO con le dovute modifiche ai vari define, macro ed header.
Risulta ora interessante valutare, anche per quanto concerne IAR PowerPac, da quali istruzioni sia composto l'entry point del nostro programma che è contenuto nel file `OS_Start_LEDBlink.c` visibile nel listato sottostante:

```
int main(void)
{

  OS_IncDI();                       /* Initially disable interrupts    */
  OS_InitKern();                    /* initialize OS                   */
  OS_InitHW();                      /* initialize Hardware for OS      */
  BSP_Init();                       /* initialize Board             */

  /* You need to create at least one task before calling OS_Start() */

  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);

  OS_Start();                      /* Start multitasking            */

  return 0;

}
```

Come si può notare dalle prime righe del `main()` sono presenti diversi comandi di inizializzazione che hanno lo scopo di avviare uno alla volta differenti frammenti utili del sistema hardware e software a partire dal kernel del sistema operativo sino alle porte led.
A questo punto risulta necessaria, trattandosi di un cambio di contesto, la creazione di almeno (per quanto ci riguarda sono più che sufficienti) due tasks ciascuno dei quali creato dalla relativa funzione. In ultimo luogo troviamo l'istruzione `OS_Start()` la quale provoca l'avvio del sistema in modalità *multitasking* e, come nel caso precedente, da questa funzione, il programma non deve più uscirne mentre, per quanto riguarda l'istruzione `return 0`, essa serve ad individuare un'eventuale esecuzione errata del programma.
Ma il vero problema è stato, anche in questa occasione, individuare l'effettiva routine del cambio di contesto; quest'ultima dopo numerose ricerche, è stata localizzata in un file di libreria denominato `OSKern.c` che risiede in una zona diversa dai files dell'applicazione.
Più propriamente la funzione interessata è `OS_INTERWORK void OS_ChangeTask(void)` la quale opera con istruzioni a più basso livello ed individua un codice particolarmente conciso ed ottimizzato; vediamo dunque di seguito la porzione di codice interessata alla valutazione sperimentale del CST su questo RTOS:

```
OS_INTERWORK void OS_ChangeTask(void) {

  GPIOInit( 4, FAST_PORT, DIR_OUT, GPIO_SC);

  OS_U8 Stat;
  OS_TASK * pActiveTask;

  OS_DICnt = 0;

  // Perform some sanity checks to find possible bugs in the assembly part

                                    [...]


#if OS_SUPPORT_SAVE_RESTORE_HOOK

                                    [...]

#endif

  //
  // For round-robin task, store the number of remaining TimeSlices
  // and handle it.
  //

  GPIO_SC_FIO ^= GPIO_SC; //semino-luciani
  GPIO_SC_FIO ^= GPIO_SC; //semino-luciani

#if OS_RR_SUPPORTED

                                    [...]
```

```
                                    [...]


#endif

                                    [...]


    //
    // Call the optional "restore context" hook function, which will typically
    // restore whatever it has saved before.
    //

  GPIO_SC_FIO ^= GPIO_SC; //semino-luciani
  GPIO_SC_FIO ^= GPIO_SC; //semino-luciani

    #if OS_SUPPORT_SAVE_RESTORE_HOOK

                                    [...]

    #endif

                                    [...]

    OS_pCurrentTask = NULL;
    OS_PREPARE_STACKPOINTER();
    OS_RegionCnt = 0;                       // OS_DICnt already =0
    OS_EI_KERNEL();
    OS_IDLE();
  }

}
```

Già dalle prime righe possiamo individuare la funzione di inizializzazione del GPIO che, oltre ad essere la stessa utilizzata per il RTOS precedente, contiene anche gli stessi parametri affinché si possa pilotare il medesimo GPIO e di conseguenza si possa accendere sempre lo stesso led. Come si può notare successivamente, le istruzioni sono molte ma sono la totalità di ciò che si rende necessario affinché si verifichi un effettivo cambio di contesto; questo, nonostante possa far apparire il programma notevolmente lungo, non deve far supporre che il costo computazionale del cambio di contesto sia maggiore in quanto è bene ricordare che, la porzione di codice considerata per il calcolo del CST al RTOS precedente, risultava essere una chiamata a funzione la quale conteneva poi a sua volta ulteriori istruzioni e chiamate, al contrario del caso attuale in cui il codice realmente eseguito è espresso *inline* direttamente all'interno della funzione principale.
Scorrendo il listato soprastante si possono notare numerose interruzioni individuate dalla presenza del simbolo `[...]`; questo sta a significare l'elisione di una porzione di codice che sarebbe comunque stata e lunga e di scarsi contenuti sperimentali. Da qui segue che il codice citato è lo stretto sufficiente alla discussione della nostra applicazione di test.

Analogamente al RTOS precedente, si può notare la presenza delle due coppie di istruzioni **`GPIO_SC_FIO ^= GPIO_SC`** poste agli estremi delle linee di codice che operano l'effettivo cambio di contesto e gestiscono l'attività multitasking; anche in questo caso la funzionalità di queste coppie è la medesima del RTOS precedente ovvero la generazione di picchi atti ad essere rilevati all'oscilloscopio ufficializzando così il passaggio del programma attraverso quelle righe e di conseguenza il tempo che intercorre tra le due.

Un'ulteriore piccola nota va spesa a favore della coerenza mantenuta nell'inserimento delle istruzioni di valutazione nei due sistemi operativi analizzati; più specificamente possiamo vedere come sia stata esclusa la funzione `OS_SUPPORT_SAVE_RESTORE_HOOK` in maniera tale da isolare, come in FreeRTOS, le pure operazioni atte al cambio di contesto scremandole così dalle più onerose e per altro simili operazioni di salvataggio e recupero dello stack di esecuzione.

A questo punto la fase di modifica del codice a scopi sperimentali si può considerare conclusa; analizzeremo, in fine, quali siano stati di fatto gli ultimi accorgimenti necessari al raggiungimento dell'esecuzione completa del programma da parte del nostro μC.

Il primo passo di questa fase finale è stata la ricompilazione delle librerie di PowerPac in quanto, il corretto funzionamento di tale sistema operativo, prevede l'utilizzo di librerie precompilate; questo *rebuild*, si è reso necessario poiché il file di nostro interesse, ovvero `OSKern.c`, si trovava all'interno di una di queste librerie e quindi non sarebbe stata sufficiente la sola modifica del file ma si rendeva invece necessario il rebuild dell'intera libreria.

La sequenza delle operazioni è stata alquanto laboriosa ed articolata; dopo aver, appunto, modificato il codice nativo nel file `OSKern.c`, abbiamo ricompilato tutto il progetto di libreria contenuto nel file di workspace `BuildLib.eww` (l'estensione `*.eww` indica un file contenente tutte le direttive per l'apertura e la modifica di un workspace di IAR Embedded Workbench), dopodichè abbiamo avviato il *file di batch* (file contenente una sequenza di comandi mandato in esecuzione dall'interprete dei comandi stesso come ad esempio lo è stato per noi MS-DOS) `BuildLibs.BAT` che, dopo la sua esecuzione, ha prodotto tutti i file ricompilati e, per quanto ci riguarda, i due file `os4t_al__d.a` (versione debug) e `os4t_al__r.a` (versione release) contenenti l'effettivo risultato della ricompilazione relativo alla porzione di codice da noi modificata per scopi sperimentali.

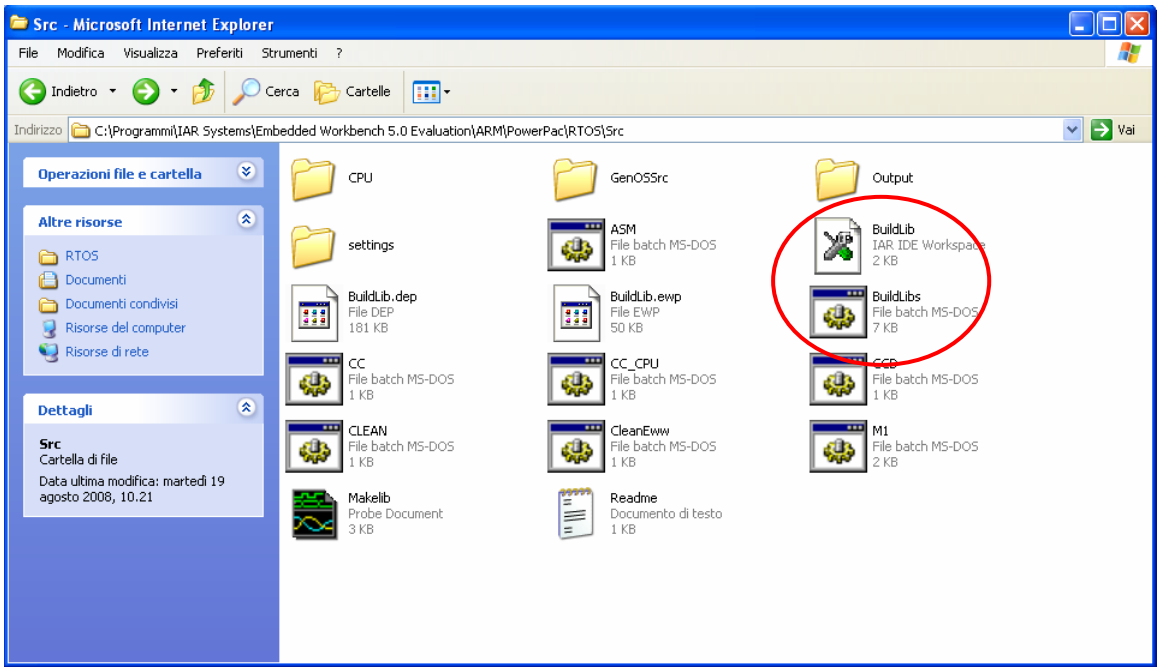


Fig. 5.4 Screenshot della cartella contenete i due file di interesse

A questo punto siamo tornati al programma principale nell'ambiente di sviluppo ed abbiamo reso noto a quest'ultimo l'esistenza delle librerie precompilate da noi aggiornate. Successivamente abbiamo ricompilato il nostro progetto dopo l'importazione delle nuove librerie e, tramite gli strumenti di debug, abbiamo "uploadato" il tutto sulla memoria Flash del nostro microcontrollore.

I collegamenti all'oscilloscopio sono rimasti invariati rispetto a quelli utilizzati per le valutazioni al RTOS precedente; anche in questo caso ciò che si è dovuto individuare sull'uscita è stato lo spazio interposto tra i due picchi generati dalle nostre istruzioni e di conseguenza il tempo che intercorre tra i due.

Per questo sistema operativo abbiamo trovato, tra i progetti forniti dal produttore, una serie di interessanti esempi tra cui è risultata di estremo interesse ed importanza una funzione che, effettuando un benchmark e quindi operante a livello software, potesse calcolare con una precisione ragionevole quale fosse l'effettivo tempo impiegato per effettuare un cambio di contesto da parte dello scheduler del sistema operativo.

Alla pagina seguente possiamo osservare il listato contenente la porzione di programma che implementa le funzionalità atte ad effettuare tale benchmark:

```
#include "RTOS.h"
#include "stdio.h"

static OS_STACKPTR int StackHP[128], StackLP[128];  // Task stacks
static OS_TASK TCBHP, TCBLP;                        // Task-control-blocks
static OS_U32 _Time;

static void HPTask(void) {
  while (1) {
    OS_Suspend(NULL);              // Suspend high priority task
    OS_Timing_End(&_Time);         // Stop measurement
  }
}

static void LPTask(void) {
  OS_U32 MeasureOverhead;          // Time for Measure Overhead
  OS_U32 v;                        // Real context switching time

  //
  // Measure overhead for time measurement so we can take this into account by
  subtracting it
  //

  OS_Timing_Start(&MeasureOverhead);
  OS_Timing_End(&MeasureOverhead);

  //
  // Perform measurements in endless loop
  //

  while (1) {
    OS_Delay(100);                 // Syncronize to tick to avoid jitter
    OS_Timing_Start(&_Time);       // Start measurement
    OS_Resume(&TCBHP);             // Resume high priority task to force task

 Switch

  // Calculate real context switching time (w/o measurement overhead)

    v = OS_Timing_GetCycles(&_Time) - OS_Timing_GetCycles(&MeasureOverhead);

  // Convert cycles to nano-seconds, increase time resolution

    v = OS_ConvertCycles2us(1000 * v);

  // Print out result

printf("Context switch time: %u.%.3u usec\r\n", v / 1000, v % 1000);

  }
}
```

Come si può vedere dal listato, oltre alle comuni righe di intestazione si possono osservare, all'interno delle funzioni relative alla creazione di processi, diverse linee di codice dedicate alla gestione di parametri e funzioni temporali; da notare come queste istruzioni di misurazione temporale operino a livello dei cicli di sistema e convertano solo

successivamente il risultato finale in µs effettuando così un `printf` sul Terminal I/O dell'ambiente di sviluppo utilizzato.
In questo contesto, non andremo ad analizzare il codice ma lo postiamo solamente qualora possa tornare utile a chi, come nel nostro caso, volesse avere un riscontro dei risultati ottenuti attraverso una via alternativa ma pur sempre attendibile.
Ecco qua calcolato, anche per IAR PowerPac, il relativo Context Switch Time.

5.1.3. Report risultati ottenuti

Come accennato al paragrafo 3.4, questo oscilloscopio è fornito dell'interessante funzione atta a catturare immagini di ciò che si propone a video in tempo reale e, tramite una fornita consolle di cursori abbinata a diverse funzioni matematiche precaricate, permette di effettuare diverse operazioni. Per quanto ci riguarda quella di maggior interesse si è rivelata la sottrazione sull'asse dei tempi così da poter calcolare, con estrema precisione, quale fosse la distanza tra i due picchi in uscita provocati da un cambio di contesto.
Vediamo qua sotto le istantanee relative a FreeRTOS, prima, e a IAR PowerPac, dopo:

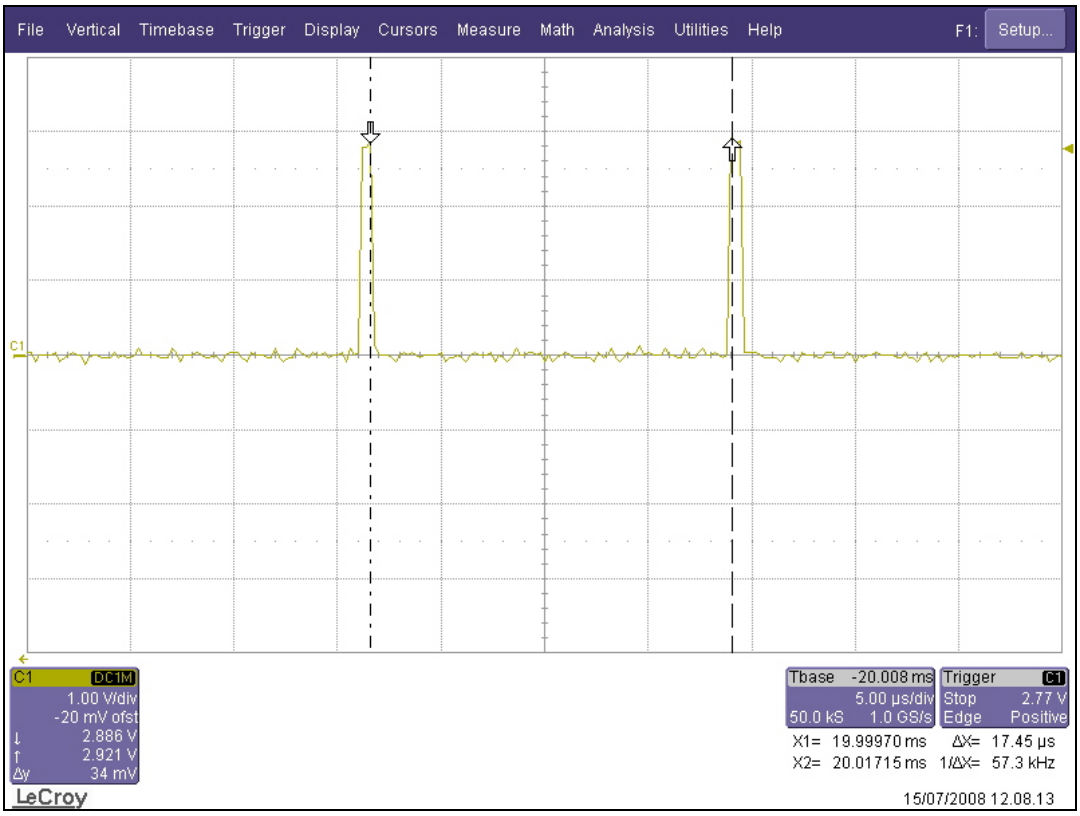


Fig. 5.5 Screenshot all'oscilloscopio del CST di FreeRTOS al primo istante

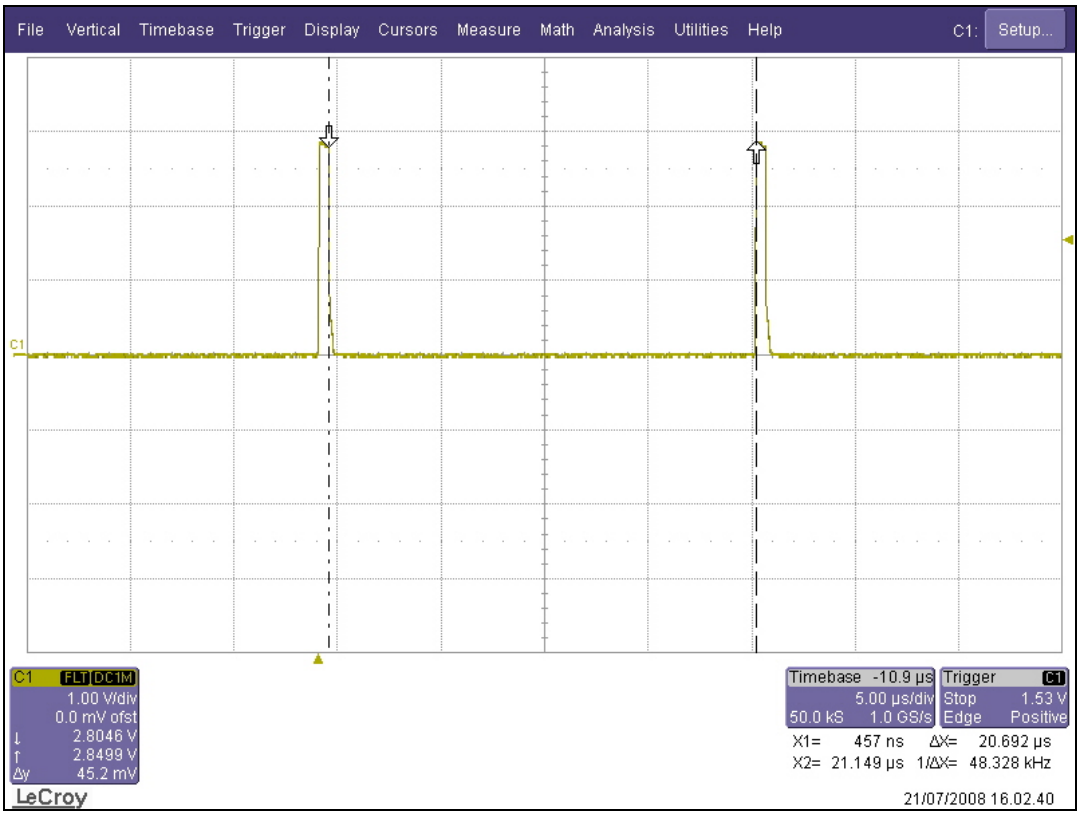


Fig. 5.6 Screenshot all'oscilloscopio del CST di FreeRTOS al secondo istante

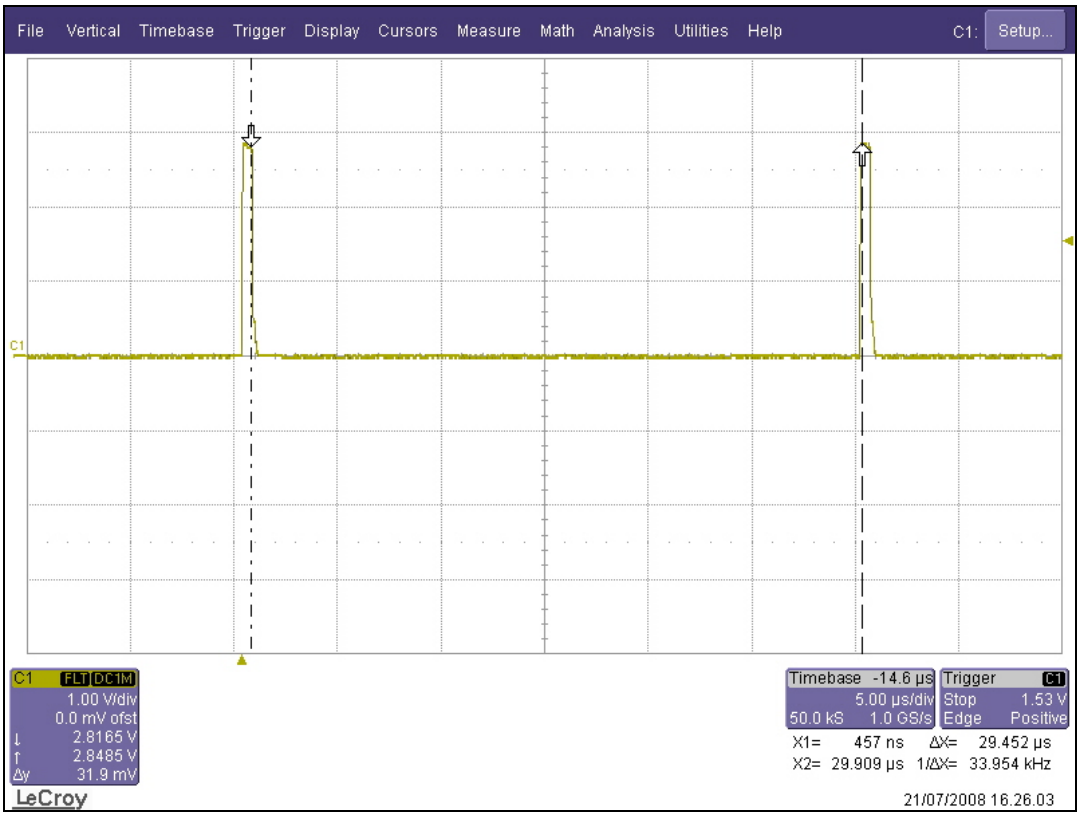


Fig. 5.7 Screenshot all'oscilloscopio del CST di FreeRTOS al terzo istante

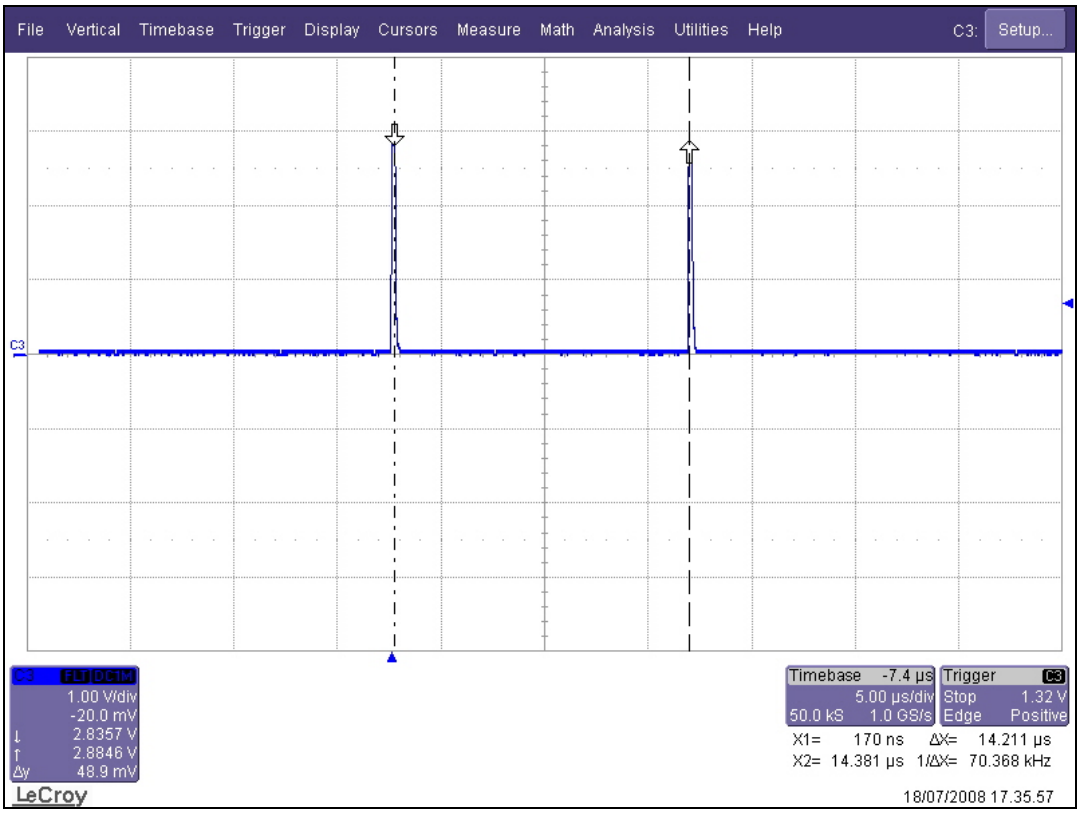


Fig. 5.8 Screenshot all'oscilloscopio del CST di IAR PowerPac

Come si può evincere dalle figure precedenti, i valori rilevati non sono assolutamente simili; infatti mentre abbiamo un unico valore stabile per IAR PowerPac, per FreeRTOS abbiamo ottenuto diversi valori in quanto, catturato in istanti diversi, mostrava tre valori differenti.

Infine è da rendere noto come sia stata particolarmente utile la verifica effettuata, attraverso l'algoritmo d'esempio fornito dall'ambiente di sviluppo, sul valore riscontrato tramite oscilloscopio per quanto riguarda il CST di IAR PowerPac; tale valore, infatti, è risultato pressoché identico ovvero 14.333 µs.

Come accennato ad inizio paragrafo, quanto appena detto è da ritenersi puramente informativo e completamente privo di qualsiasi commento; ci occuperemo in seguito nel paragrafo 6.1.1 di analizzare quanto più approfonditamente i risultati ottenuti commentandoli in maniera tecnicamente critica.

5.2. Interrupt Latency

Nel paragrafo 2.2.4 abbiamo già discusso riguardo il significato di Interrupt Latency ovvero abbiamo individuato in questa parola il tempo che intercorre tra una richiesta di interrupt e l'inizio dell'esecuzione del processo relativo a tale interruzione.
La tecnica operativa è stata analoga a quella attuata nel caso del calcolo del CST ovvero abbiamo proceduto a piccoli passi verificando di volta in volta la correttezza, o l'eventuale scorrettezza, dei passaggi operativi.

5.2.1. Criterio sperimentale

Ci troviamo ora di fronte ad un nuovo problema al quale bisognerà abbinare, se possibile, una nuova soluzione; bisogna quindi applicare un criterio sperimentale che ci porti ad ottenere il risultato preposto.
Se nel caso del CST l'input era totalmente generato via software dallo scheduler del sistema operativo, ci troviamo ora invece a dover fare i conti con la generazione di un input prodotto all'esterno da noi stessi. Quindi il primo passo da compiere e, se vogliamo anche il più laborioso in questo contesto, è stato quello, non solo di progettare un piccolo circuito in grado di generare un'interruzione, ma anche il dover configurare tutti i parametri necessari, a livello di sistema operativo, per praticare l'acquisizione del segnale di interrupt esterno.
Per quanto riguarda l'uscita, la questione è stata ben più semplice in quanto abbiamo riutilizzato la già collaudata funzione atta al pilotaggio di un LED tramite GPIO.
A questo punto non resta che misurare, per mezzo dell'oscilloscopio, quale sia l'effettivo tempo che intercorre tra la variazione di livello generata dal circuito in ingresso e l'accensione del LED prodotta in uscita dalla routine abbinata a tale interrupt.

5.2.2. Algoritmo implementato

Prima di addentrarci completamente nella descrizione di ciò che è stato effettivamente svolto per portare a termine il calcolo dell'Interrupt Latency, relativamente ai due sistemi, è bene fare un'importante, quanto fondamentale, premessa. Dovendo modificare, con opportune istruzioni di test, il codice sorgente di entrambi i sistemi operativi, ci siamo posti

il problema di dove risiedessero le funzioni, o più precisamente, le linee di codice dell'*Interrupt Handler* ovvero del gestore di interrupt.

Con molta sorpresa da parte nostra e, dopo un'assidua ricerca passiva in rete ed attiva su forum specifici all'argomento [28], abbiamo scoperto che il sistema FreeRTOS è privo, almeno per quanto riguarda la nostra architettura specifica, di un gestore degl'interrupt ed è quindi consigliata l'importazione di tale porzione di sistema operativo da terze parti effettuando, chiaramente, le opportune rettifiche di adattamento.

Per quanto riguarda IAR PowerPac la gestione degli interrupt è totalmente integrata in alcune funzioni di libreria.

Il codice relativo all'*interrupt handling* di PowerPac individua una gestione delle interruzioni piuttosto performante; con questa premessa siamo convenuti, data la disponibilità di FreeRTOS ad accettare inclusioni di codice importato da terzi, nel riutilizzare la medesima porzione di codice di PowerPac anche per FreeRTOS.

Quanto detto sopra giustifica il fatto che questo paragrafo, a differenza del 5.1.2, non sia bipartito tra i due RTOS in quanto, per entrambi i sistemi, il codice analizzato e successivamente modificato è lo stesso e quindi si renderebbe superficiale ed inutile un'analisi separata facendo essi stessi, di fatto, capo alla stessa fonte.

Come per il calcolo del CST, abbiamo dovuto individuare alcuni pins che potessero essere configurati come uscite (in questo caso abbiamo utilizzato gli stessi); in questo contesto abbiamo, inoltre, dovuto cercare dei pins configurabili come ingresso per il nostro microcontrollore in maniera tale da abbinarvi un interrupt.

Tutto ciò è stato fatto sulla base di poter usufruire di un connettore sull'Evaluation Board che potesse essere comodamente raggiungibile con una sonda in maniera tale da poter verificare il corretto funzionamento del circuito generatore di interrupt.

La prima cosa che abbiamo quindi fatto è stata individuare sul manuale dell'utente dell'LPC23xx [27] quali fossero gli effettivi pins in grado di acquisire un segnale elettrico dall'esterno considerandolo una richiesta di interrupt (IRQ); di pins adatti al nostro scopo ve ne sono ben quattro ma dovendo sceglierne uno solo consideriamo il primo: il P2.10.

È interessante notare come il P2.10, come del resto ognuno di questi pins, possa avere funzionalità differenti a seconda del suo valore di configurazione; tale valore viene definito attraverso la modifica dei bit 21 e 20 all'interno del registro a 32 bit **`PINSEL4`** visibile nella tabella alla pagina seguente:

REGISTRO: PINSEL4															
31:30	29:28	27:26	25:24	23:22	**21:20**	19:18	17:16	15:14	13:12	11:10	9:8	7:6	5:4	3:2	1:0
P2.15	P2.14	P2.13	P2.12	P2.11	**P2.10**	P2.9	P2.8	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0

Tab. 5.5 Astrazione schematica del registro PINSEL4

A questo punto non rimane che settare i suddetti bit rispettivamente a 0 e 1 così da configurare il pin P2.10 in modalità $\overline{EINT0}$ ovvero come accettore di interrupt esterni.
Successivamente, dopo aver consultato lo schema elettrico dell'EB (Tavola n° 1), abbiamo notato che il suddetto pin è direttamente collegato al terminale 24 della porta EXT2.
Come visibile in Fig. 5.10, abbiamo necessitato dell'ausilio di ulteriori due pins EXT2-39 ed EXT2-40 connessi di default rispettivamente ai valori notevoli di tensione quali la alimentazione (3.3V) e la massa (0V) così da alimentare il circuito generatore di interrupt.
Definiti i collegamenti fisici dei pins e le loro modalità di funzionamento abbiamo progettato, grazie alla collaborazione dell'Ing. Luigi Veardo, un piccolo circuito elettrico ad hoc in grado di provocare una variazione di livello in termini di tensione ai capi di due pins di cui, uno è necessariamente EXT2-24 mentre l'altro, a seconda dello stato in cui si trova il circuito, può essere o l'EXT2-39 o l'EXT2-40; tale variazione di tensione verrà dunque interpretata come un interrupt da parte del μC.
Il circuito elettrico per la generazione di interrupt da noi utilizzato è così costituito:

- Contatto normalmente aperto
- Una resistenza di *pull-up* da 11Ω
- Uno zoccolo nero compatibile con la porta EXT2

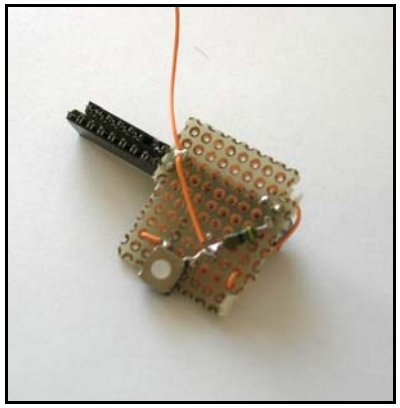


Fig. 5.9 Dispositivo generatore di Interrupt

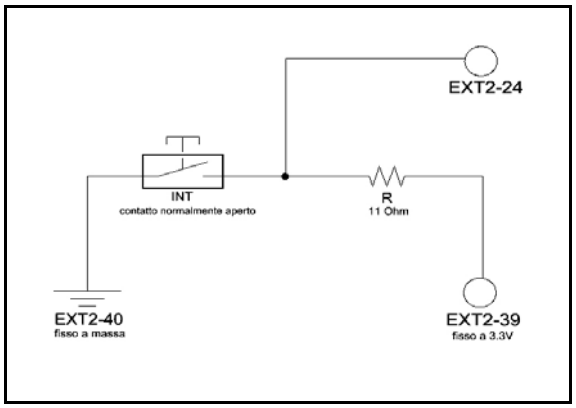


Fig. 5.10 Schema circuitale del generatore di Interrupt

Volendo fare una breve descrizione di questo semplice circuito, sulla base dei componenti e dello schematic alla pagina precedente, possiamo individuare come il valore di tensione letto dal pin EXT2-24 sia direttamente dipendente dall'azione di pressione del microinterruttore; tale azione deve per forza essere effettuata da noi e l'interrupt sarà quindi generato dall'esterno.

Per quanto riguarda il circuito di notifica dell'output, ovvero il LED, è stato mantenuto quello progettato per il calcolo del CST (par. 5.1.2) con i medesimi collegamenti sull'STK.

Veniamo ora al sodo ovvero ci addentriamo nel codice che ha permesso di mettere in relazione l'interrupt in ingresso con l'accensione del led in uscita.

L'idea di partenza è stata quella di mettere in condizioni il sistema operativo di prendere in consegna un interrupt esterno ed avviare la relativa routine di handling; per fare ciò il primo file con cui abbiamo avuto a che fare è stato `target.c` e, più specificamente, abbiamo dovuto modificare il registro **`PINSEL4`** presente nel corpo della funzione:

```
void GPIOResetInit( void )
{
  /* Reset all GPIO pins to default: primary function */

  PINSEL0 = 0x00000000;
  PINSEL1 = 0x00000000;
  PINSEL2 = 0x00000000;
  PINSEL3 = 0x00000000;

  PINSEL4 = 0x00100000;     //semino-luciani
  //PINSEL4 = 0x00000000;   //semino-luciani

  PINSEL5 = 0x00000000;
  PINSEL6 = 0x00000000;
  PINSEL7 = 0x00000000;
  PINSEL8 = 0x00000000;
  PINSEL9 = 0x00000000;
  PINSEL10 = 0x00000000;

  IO0DIR = 0x00000000;
  IO1DIR = 0x00000000;

  FIO0DIR = 0x00000000;
  FIO1DIR = 0x00000000;
  FIO2DIR = 0x00000000;
  FIO3DIR = 0x00000000;
  FIO4DIR = 0x00000000;

  FIO0MASK = 0x00000000;     /* not used */
  FIO1MASK = 0x00000000;     /* not used */
  FIO2MASK = 0x00000000;     /* not used */
  FIO3MASK = 0x00000000;     /* not used */
  FIO4MASK = 0x00000000;     /* not used */

  return;
}
```

Come già visto per definire le modalità di funzionamento del pin P2.10, bisogna operare sul registro **PINSEL4** che è espresso in base esadecimale. Poiché ogni valore esadecimale può essere espresso con quattro bit e dovendo noi andare a modificare i bit 21 e 20 di tale registro a 32 bit abbiamo operato la seguente conversione:

```
BIN: 0000 0000 0001 0000 0000 0000 0000 0000
HEX: 0x00100000
```

Un altro file che abbiamo dovuto modificare per raggiungere l'obiettivo preposto è stato `einttest.c` contenete, oltre ad una serie di include e di define, solamente l'entry point:

```
[...]

#define GPIO_SC  (1UL<<0)

[...]

//ENTRY POINT
int main (void)
{
    TargetResetInit();

    printf("START PROGRAM\n"); //semino-luciani

    GPIOInit( 4, FAST_PORT, DIR_OUT, GPIO_SC);
    SCS_bit.GPIOM = 1;                      /* enable fast io for GPIO0,1 */
    USB_LINK_LED_FDIR = USB_LINK_LED_MASK;  /* USB Link LED port output */
    USB_LINK_LED_FSET = USB_LINK_LED_MASK;  /* turn off USB Link LED */

    /* initialize GPIO pins as external interrupts */
    EINTInit();

    /****** It's an endless loop waiting for external interrupt ******/
    /* EINT3 can be used to test the external interrupt
      It's shared with GPIO0,2 interrupts */
    while( 1 )
    {
    // printf("INTERRUPT OK %x!!!!!\n",eint0_counter); //semino-luciani
    }

    return 0;
}
```

Scorrendo il listato le prime istruzioni che troviamo sono quelle relative alle operazioni di inizializzazione tra cui, le più importanti al nostro scopo, sono due: la prima è `TargetResetInit()` che contiene le chiamate alla funzione di inizializzazione del GPIO considerato come interrupt in ingresso ed alla funzione relativa all'inizializzazione della VIC; successivamente, nell'entry point, troviamo la chiamata alla funzione denominata

`GPIOInit( 4, FAST_PORT, DIR_OUT, GPIO_SC )` che è relativa al pilotaggio del GPIO per quanto riguarda la gestione dell'uscita.

A questo punto il programma esegue l'istruzione `EINTInit()` che è definita nel file `extint.c`; buona parte delle modifiche apportate al codice sono state effettuate all'interno di questa funzione e vale quindi la pena di analizzarle singolarmente:

```
DWORD EINTInit( void )
{

  EXTMODE |= 1;           //semino-luciani
  EXTPOLAR |= 1;          //semino-luciani

  IO0INTENF = B1_MASK;    /* B1 interrupt by falling edge. */

  //semino-luciani

  if (install_irq(EINT0_INT, (void *)EINT0_Handler, HIGHEST_PRIORITY) == FALSE)

  //semino-luciani

  {
    return (FALSE);
  }

  return( TRUE );
}
```

Se prima abbiamo inizializzato il P2.10 come interrupt, definiamo ora le modalità con cui esso debba essere rilevato e successivamente installato nella tabella delle interruzioni; per agevolare la valutazione dei risultati all'oscilloscopio, abbiamo dovuto modificare i due registri da 8 bit ciascuno **`EXTMODE`** ed **`EXTPOLAR`** visibili nelle due tabelle seguenti:

REGISTRO: EXTMODE							
7	6	5	4	3	2	1	**0**
RESERVED	RESERVED	RESERVED	RESERVED	EXTMODE3	EXTMODE2	EXTMODE1	**EXTMODE0**

Tab. 5.6 Astrazione schematica del registro EXTMODE

REGISTRO: EXTPOLAR							
7	6	5	4	3	2	1	**0**
RESERVED	RESERVED	RESERVED	RESERVED	EXTPOLAR3	EXTPOLAR2	EXTPOLAR1	**EXTPOLAR0**

Tab. 5.7 Astrazione schematica del registro EXTPOLAR

Sulla base delle informazioni reperite sullo User Manual dell'LPC23xx [27], abbiamo visto che, per poter configurare le modalità di rilevazione degli interrupt da noi scelte relativamente al pin P2.10 in qualità di $\overline{EINT0}$,è necessario porre i bit meno significativi di entrambi i registri a 1. Per quanto riguarda il primo registro, abbiamo settato il relativo LSB a 1 in maniera tale che il gestore degli interrupt fosse *edge sensible* ovvero sensibile ai fronti piuttosto che ai livelli; per quanto riguarda il secondo, esso va ad affinare ulteriormente la sensibilità circoscrivendo la stessa prettamente nell'individuazione dei fronti di salita.

Come si può notare dalle istruzioni in maggior evidenza del codice, vi è la presenza dell'operatore **=|** che effettua l'operazione logica di OR bit a bit tra due variabili di uguale lunghezza quali, nel nostro caso, ciascuno dei due registri con una maschera adeguata.

Poiché ogni bit di questi registri svolge una ben definita funzione che può essere anche relativa a contesti funzionali diversi da quello in cui si sta attualmente operando, eccetto il bit realmente coinvolto nelle operazioni interessate, è di fondamentale importanza che le modifiche a ciascuno di tali registri interessino solo e soltanto il bit direttamente interessato e lascino inalterata la configurazione dei bit ad esso complementari nel registro.

Per il raggiungimento di questo scopo ci viene direttamente incontro l'operazione sopraccitata di OR logico visibile nella tabella seguente:

OR		
X	**Y**	**U**
0	0	0
0	1	1
1	0	1
1	1	1

Tab. 5.8 Tabella di verità dello OR

Considerando il segnale X come controllo si può rilevare questa interessante proprietà:

```
(X = 0) ➔ U = Y
(X = 1) ➔ U = 1
```

Relativamente a questa proprietà si può notare che applicando il controllo X pari a 0 e l'ingresso Y, in uscita otteniamo il valore di Y stesso; diversamente se poniamo il controllo X a 1 e diamo nuovamente come ingresso Y, in uscita otterremo sempre 1.
In questa maniera, introducendo una maschera adeguata del tipo 0x1, riusciremo a modificare l'LSB di tali registri al valore preposto 1 lasciando comunque invariato lo stato dei rimanenti bit non interessati costituenti il registro stesso.
L'ultima istruzione importante di questa porzione di codice è la chiamata alla funzione `DWORD install_irq( DWORD IntNumber, void *HendlerAdrr, DWORD Priority )` la quale è utile ad installare ed allocare la nostra IRQ all'interno della VIC; al momento dell'installazione di una nuova IRQ, quest'ultima, verrà inserita in una locazione libera nella tabella delle interruzioni.
Da non tralasciare è il fatto che i due registri **`EXTMODE`** ed **`EXTPOLAR`** vadano a costituire, insieme al registro da 8 bit **`EXTINT`**, l'informazione che permetta di definire le modalità di rilevazione dell'interruzione; tale registro è visibile nella tabella seguente:

REGISTRO: EXTINT							
7	6	5	4	3	2	1	**0**
RESERVED	RESERVED	RESERVED	RESERVED	EINT3	EINT2	EINT1	**EINT0**

Tab. 5.9 Astrazione schematica del registro EXTINT

Essendo noi interessanti all'interrupt esterno $\overline{EINT0}$ andremo, come nei casi precedenti, a configurare l'LSB di tale registro contenente i flags relativi all'interrupt esterno.
Il settaggio di questo registro è visibile nella funzione `__irq __nested __arm void EINT0_Handler (void)` contenuta nel file `extint.c` per la quale vale la pena commentare il relativo codice:

```
__irq __nested __arm void EINT0_Handler (void)
{
   EXTINT |= 1;                  //semino-luciani
  __disable_interrupt();         /* handles nested interrupt */
  eint0_counter++;
  /* flashing USB Link LED */
  GPIO_SC_FIO ^= GPIO_SC;        //semino-luciani
```

```
   printf("INTERRUPT OK!!!!!\n"); //semino-luciani
   //USB_LINK_LED_FIO ^= USB_LINK_LED_MASK;
   __enable_interrupt();        /* handles nested interrupt */
   VICADDRESS = 0;              /* Acknowledge Interrupt */
}
```

La funzione precedente non è altro che il gestore degli interrupt relativamente al P2.10 in modalità $\overline{EINT0}$; la prima istruzione che incontriamo, ovvero la modifica del registro **`EXTINT`**, è estremamente importante in quanto tale registro contiene gli *Interrupt Flags* permettendo così di individuare la corrispondente richiesta d'interruzione alla VIC; anche in questo caso per poter settare il bit meno significativo di tale registro senza compromettere lo stato dei restanti bit, effettueremo un'operazione di OR bit a bit tra il registro stesso ed un'opportuna maschera del tipo 0x1.
Immediatamente dopo troviamo la chiamata per disabilitare gli interrupt in maniera tale da non avere interruzioni concorrenti durante la routine di esecuzione dell'IRQ.
La successiva riga di nostro interesse è quella atta alla variazione del livello sul pin configurato come GPIO in uscita; essa, come nel caso del calcolo del CST, effettua l'operazione di XOR atta a generare una transizione di livello dipendente dallo stato precedente.
È stata inoltre inserita, per motivi di ulteriore verifica, una `printf` con lo scopo di mostrare, sul Terminal I/O, la corretta esecuzione delle operazioni inserite nell'interrupt handler; tale soluzione è stata adottata con lo scopo di avere un riscontro più facilmente osservabile all'occhio umano e per avere così una conferma puramente qualitativa relativamente alla correttezza delle operazioni svolte.
Le ultime due istruzioni sono fondamentali in quanto riabilitano la disponibilità, da parte del sistema, ad accettare nuovamente delle interruzioni.
A questo punto è tutto pronto per il test.
Avviamo dunque la compilazione e, corretta una piccola serie di errori dovuta ad alcune sviste, inviamo il tutto alla memoria Flash del nostro microcontrollore e attiviamo il debug.
Ponendo così due sonde, una ai capi del LED ed una ai capi del microinterruttore (ovvero tra il pin EXT2-24 ed il pin EXT2-40), è possibile finalmente individuare il tempo che intercorre tra il fronte che individua la pressione del microinterruttore ed il fronte che individua l'esecuzione della routine di interrupt associata.

5.2.3. Report risultati ottenuti

Andiamo ora a fare una breve rassegna di quanto è stato visualizzato sul monitor dell'oscilloscopio senza però, come nel caso del report relativo al CST, trarre alcuna conclusione.

Vediamo qua sotto un'istantanea scattata dopo aver posizionato, tramite la consolle dei cursori, i riferimenti utili ad effettuare l'operazione di sottrazione sull'asse dei tempi tra i due fronti volutamente mutati in ordine complementare in maniera tale da esaltare lo spazio che di fatto li separa ovvero l'Interrupt Latency.

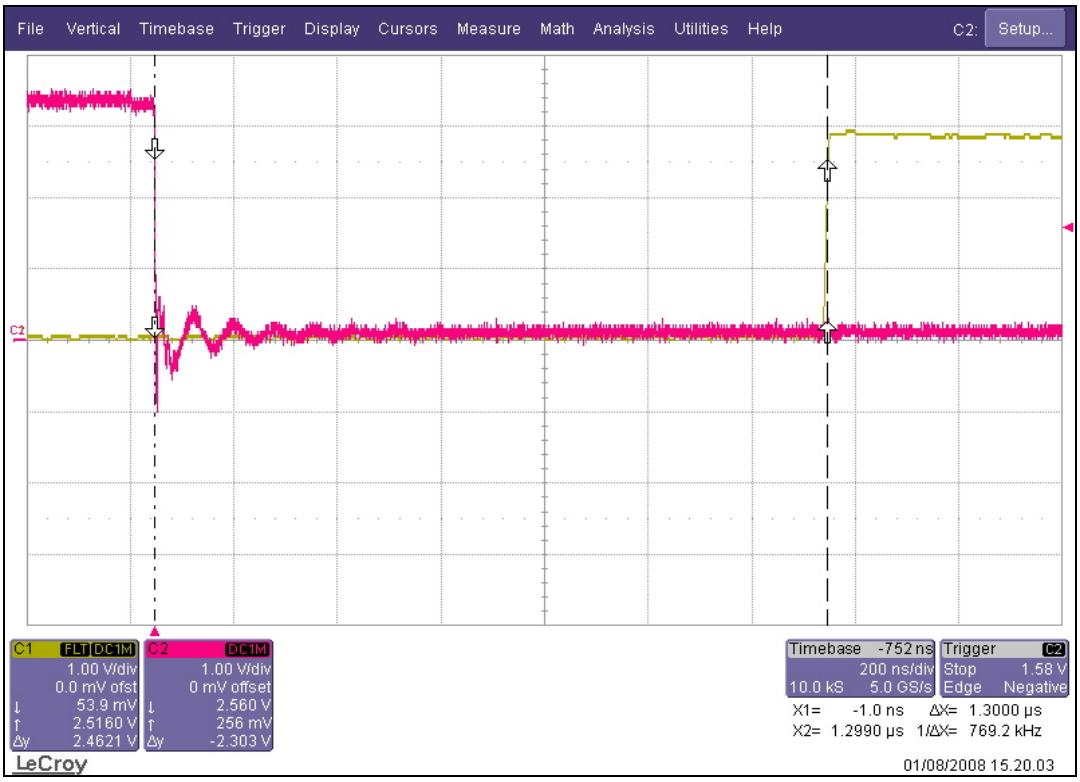


Fig. 5.11 Screenshot all'oscilloscopio dell'Interrupt Latency di IAR PowerPac

Come accennato all'inizio del paragrafo 5.2.2, il risultato è da considerarsi coincidente tra i due sistemi per i motivi già specificati in precedenza; da qui andremo a commentare e quindi valutare, al paragrafo 6.1.2, tale misurazione effettuando le dovute considerazioni a riguardo.

5.3. Footprint

Prima di discutere riguardo l'analisi del Footprint relativo ai due RTOS è bene effettuare, riprendendo quanto detto al paragrafo 2.2.2, alcuni richiami teorici in maniera tale da porre sufficienti basi al successivo confronto.
Di fatto il Footprint definisce l'ammontare di memoria a cui il programma si riferisce durante la sua esecuzione; chiaramente il Footprint include, nel suo significato, tutte quelle regioni di memoria attive che possono contenere codice, stack, heap, costanti, strutture di debug, tavole di simboli, etc necessari durante l'esecuzione del programma le quali vengono caricate almeno una volta.
Un programma di grosse dimensioni avrà un Footprint particolarmente ingombrante; altresì qualora avessimo a che fare, come nel nostro caso, con un dispositivo la cui memoria è particolarmente risicata e con funzionalità ben più ridotte di un calcolatore da tavolo, il Footprint, oltre che a essere più ridotto, sarà anche circoscritto ad un tipo di dati più semplici quali codice, costanti e variabili.
Passiamo ora alla descrizione del percorso che ci ha condotti al reperimento dei dati utili.

5.3.1. Criterio sperimentale

In questo caso, effettivamente, il percorso si è mostrato ben meno tortuoso di quanto non lo sia stato per il calcolo dei due precedenti parametri; infatti, ogniqualvolta venga compilato il progetto identificante l'intero programma, all'interno di una cartella relativa agli output viene prodotto, in maniera del tutto automatica, un file `*.map` contenente un resoconto dettagliato di quale sia stata, relativamente ad ogni singolo file, la propria occupazione in memoria fornendo inoltre valori specifici e totali ed informazioni riguardo la posizione in memoria delle funzioni "linkate" nella nostra applicazione.

5.3.2. Algoritmo implementato

Come nel caso precedente si può notare che è stata omessa la bipartizione individuante le differenze operative effettuate per raggiungere lo scopo; in questo caso il motivo non è lo stesso poiché, per quanto riguarda il Footprint, i valori rilevati sono sì diversi tra i due RTOS ma, non essendoci alcun algoritmo da modificare e non dovendo specificare quindi

eventuali strade perseguite al raggiungimento dello scopo, ci siamo limitati alla interpretazione dei dati contenuti all'interno del file `*.map` .
Ogniqualvolta si effettui un *rebuild all* i files `testRTOS.map` per FreeRTOS e `Start_LPC2378.map` per IAR PowerPac, vengono rigenerati ed il Footprint ricalcolato.
Il file che definisce l'occupazione di memoria di FreeRTOS è contenuto all'interno del path `…\Debug\List` che a sua volta appartiene al percorso del progetto; per quanto riguarda IAR PowerPac tale file è individuabile nel percorso `…\Output\LPC2378\Debug_Flash\List` ed è anch'esso contenuto nella directory di progetto.
All'interno di ciascuno di questi due files si possono trovare svariate informazioni tra cui:

- Readonly Code Memory
- Readonly Data Memory
- Readwrite Data Memory

La prima individua l'occupazione di memoria da parte del codice sorgente che è di fatto in sola lettura mentre la seconda esprime la quantità di memoria occupata da parte di dati in sola lettura ovvero le costanti; a differenza di queste due, l'ultima componente dell'elenco individua invece quella porzione di memoria che viene utilizzata dalle variabili del programma, sia in lettura che in scrittura.
Come in ogni test fatto finora, è bene spendere alcune parole sulle ipotesi di coerenza relative al corretto svolgimento dei tests in quanto è importante notare come l'occupazione in memoria (Flash e RAM) di un RTOS possa variare dipendentemente dalla modalità di compilazione impostata:

- *Processor Mode*: questo parametro può essere impostato come ARM o Thumb; nel primo caso si avrà l'installazione del codice completo con la conseguente disponibilità di tutte le funzionalità del μC mentre il secondo comporterà una riduzione del codice installato in quanto gran parte dei registri non verrà utilizzata.
- *Debugging informations*: abilitando l'inserimento delle informazioni di debug nel file oggetto in uscita, si avrà un conseguente ed inevitabile incremento di memoria utilizzata; altresì, qualora non sia necessaria alcuna utility di debug a bordo, si può disabilitare tale opzione ottenendo così un conseguente programma più "leggero".

Per quanto riguarda le ipotesi di coerenza del nostro test abbiamo posto il compilatore in modalità ARM tralasciando l'inserimento delle informazioni di debug per entrambi i sistemi operativi messi a confronto.

5.3.3. Report risultati ottenuti

Troviamo qui di seguito la parte riassuntiva contenuta nelle ultime righe dei relativi files `*.map` riguardante il Footprint dei due sistemi ed i relativi significati numerici:

```
FREERTOS:


  16 368 bytes of readonly  code memory
     269 bytes of readonly  data memory
  20 645 bytes of readwrite data memory

Errors: none
Warnings: none
```

Fig. 5.12 Footprint di FreeRTOS

```
IAR POWERPAC:


  16 492 bytes of readonly  code memory
      80 bytes of readonly  data memory
   1 576 bytes of readwrite data memory

Errors: none
Warnings: none
```

Fig. 5.13 Footprint di IAR PowerPac

Non stiamo, come nei report precedenti, a fornire commenti riguardanti i risultati ottenuti ma diciamo soltanto che, in fase di compilazione possono esserci diverse modalità di generazione dell'output; per quanto ci riguarda abbiamo configurato il compilatore in maniera identica per entrambi i sistemi operativi cosicché le discrepanze risultanti dall'esito del test fossero dovute solamente all'effettiva gestione differente della memoria tra i due sistemi e non a ipotesi operative dissimili.

6. Analisi dei risultati e conclusioni

Eccoci giunti all'epilogo di questa interessante esperienza.
Questo capitolo si configura come compendio basato sui risultati ottenuti e consultabili nei relativi reports al Capitolo 5; effettuato tale compendio, verrà fatta un'analisi quanto più possibile approfondita degli esiti relativi ai tests eseguiti e, successivamente, verranno tratte le considerazioni finali con lo scopo di individuare in conclusione quale, tra i due RTOS messi a confronto, sia per noi effettivamente "il sistema".

6.1. Commenti ai risultati ottenuti

Ai paragrafi precedenti abbiamo visto le modalità con cui si sono svolti i tests relativi ai tre parametri caratteristici riguardanti i due RTOS messi a confronto ovvero il CST, l'Interrupt Latency ed il Footprint e successivamente abbiamo riportato, in maniera puramente informativa, i risultati ottenuti. Vediamo ora nel particolare di commentare i reports precedenti e di trarre, per ciascun parametro, una specifica conclusione relativamente ai due RTOS confrontati.

6.1.1. Context Switch Time

Come prima cosa è bene riportare in forma più schematica e comprensibile quanto citato nel report relativo alla valutazione di questo parametro nei due RTOS.
Scrutando il grafico alla pagina seguente e la relativa tabella contenenti i valori specifici, salta subito all'occhio come IAR PowerPac effettui il cambio di contesto in tempo minore assestandosi a 14,21 μs. Correlato a questo risultato è interessante notare come nelle varie sessioni notevoli di test, tale sistema, oltre che essere il più rapido tra i due, mantenga con estrema costanza il valore del suo CST.
Segue quindi l'istogramma a barre orizzontali sopraccitato e la relativa tabella contenenti i tempi di cambio di contesto di FreeRTOS e IAR PowerPac relativamente ad alcune sessioni di test notevoli che si sono rese utili, rispetto all'attuazione di un'unica sessione, alla rilevazione dei valori che li contraddistinguono:

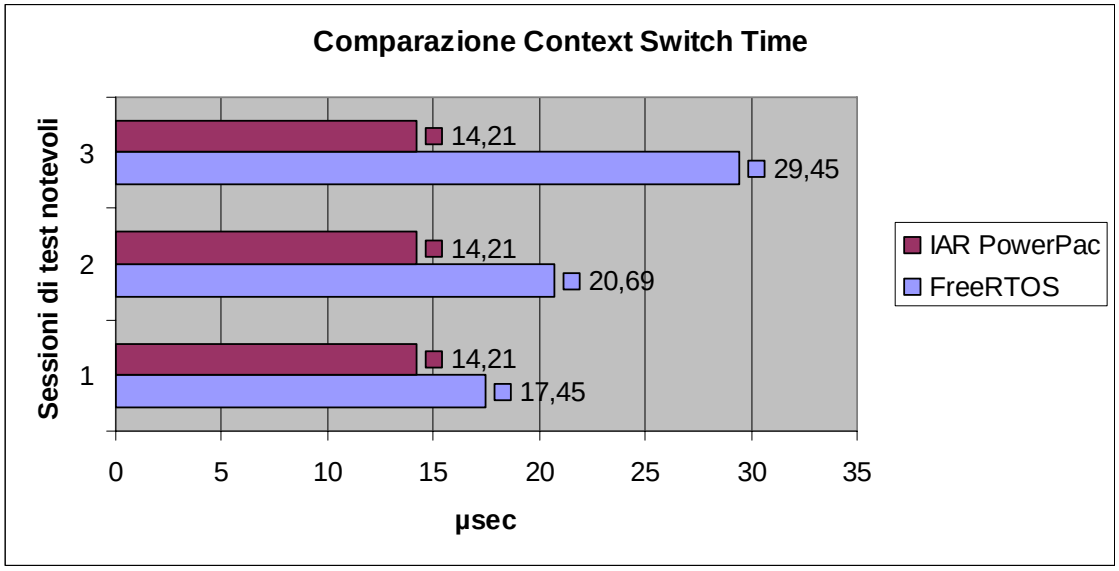


Fig. 6.1 Istogramma di comparazione tra i due RTOS relativamente al CST

SISTEMA OPERATIVO	SESSIONI DI TEST		
	prima sessione	seconda sessione	terza sessione
FreeRTOS	17,45 (µs)	20,69 (µs)	29,45 (µs)
IAR PowerPac	14,21 (µs)	14,21 (µs)	14,21 (µs)

Tab. 6.1 Valori specifici relativi al CST dei due RTOS

In maniera direttamente contrapposta a quanto detto sopra si può notare come, per quanto riguardi FreeRTOS, il tempo di cambio di contesto sia palesemente superiore denotando così sullo stesso terreno una velocità inferiore. Ma non è tutto; infatti, come è facilmente visibile dal grafico sopra, si può notare una notevole varianza del valore relativo al CST di tale sistema, campionato nelle varie sessioni di test. Qui di seguito mostriamo inoltre un istogramma rappresentante il valore medio calcolato sulle tre sessioni di test notevoli:

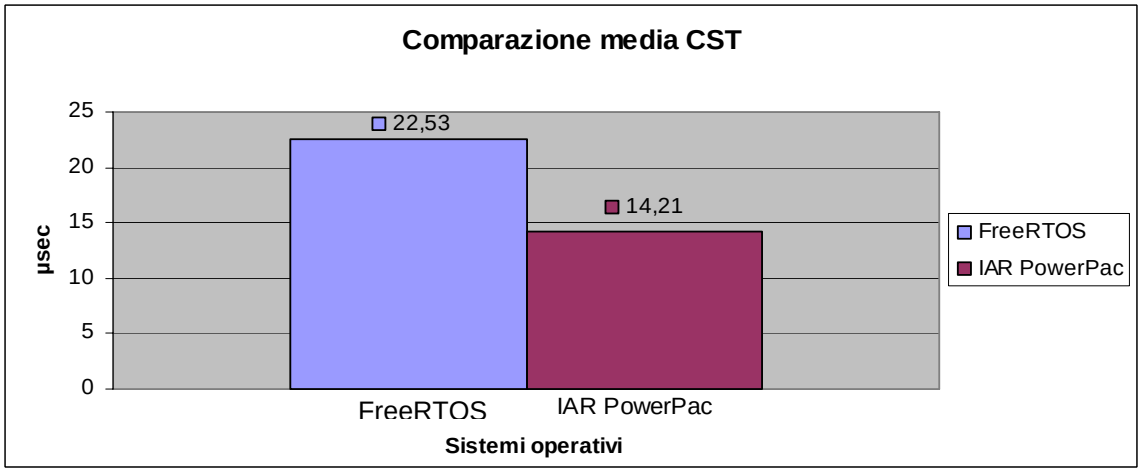


Fig. 6.2 Comparazione media del CST dei singoli RTOS

Volendo concludere commentando i risultati citati in precedenza, non si può negare come le performance di IAR PowerPac siano migliori offrendo un CST inferiore e soprattutto stabile rispetto a FreeRTOS; questo, viste le premesse di coerenza nella valutazione, è da attribuire ad una migliore gestione dello scheduler da parte di IAR PowerPac.
Da notare inoltre, come accennato al paragrafo 5.1.3, che il risultato ottenuto dalle misurazioni del CST sopra descritte relativamente a IAR PowerPac, è a maggior ragione attendibile in quanto, valutando lo stesso parametro attraverso un benchmark dedicato, risulta che lo scostamento tra i due valori, hardware e software, sia pressoché nullo.

6.1.2. Interrupt Latency

In questo caso la situazione è un po' differente dal comune in quanto, come accennato in apertura al paragrafo 5.2.2, per l'architettura da noi utilizzata, il sistema operativo FreeRTOS non dispone di un gestore degli interrupt.
Qui entra in gioco IAR PowerPac; esso infatti è provvisto di funzionalità complete atte a gestire gli interrupt con ottimi risultati. Questo ci ha portati a prendere la decisione di analizzare quindi soltanto le performance date da quest'ultimo in vista di importare, in un futuro, lo stesso gestore degli interrupt in FreeRTOS.
Volendo trarre qualche somma possiamo dire che IAR PowerPac, oltre che ad essere fornito di un Interrupt handler, quest'ultimo è anche particolarmente ottimizzato rendendo così IAR PowerPac di grande interesse relativamente ai nostri scopi applicativi.
Altresì per FreeRTOS non si può dire molto in quanto il gestore degli interrupt è del tutto inesistente per la nostra architettura.

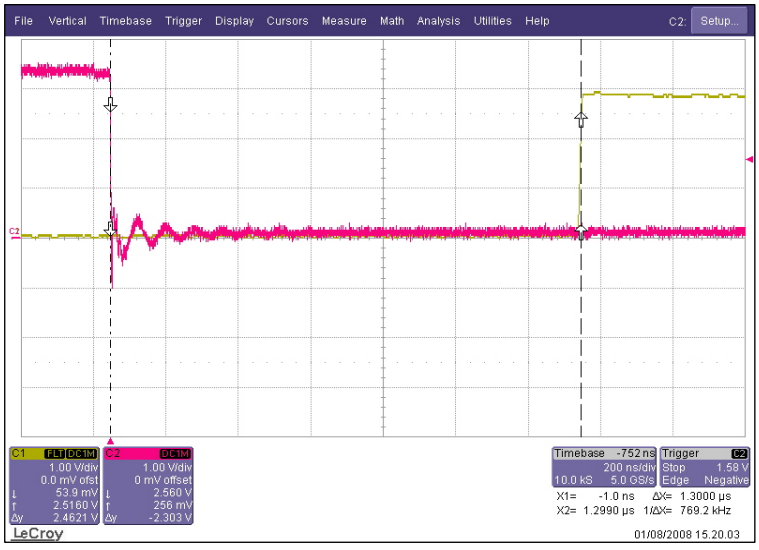


Fig. 6.3 Screenshot all'oscilloscopio dell'Interrupt Latency di IAR PowerPac

In questa occasione, non trattandosi di un vero e proprio confronto a livello numerico, abbiamo rappresentato qui sopra, piuttosto che un grafico, la schermata fornita dall'oscilloscopio relativamente all'Interrupt Latency di IAR PowerPac in maniera tale da poter apprezzare la risicatezza del suddetto valore pari a 1.3 µs.

6.1.3. Footprint

Per avere una visione generale di come e quale sia stato l'esito delle valutazioni fatte relativamente al Footprint calcolato per entrambi i RTOS, è bene innanzitutto dare uno sguardo ai grafici sottostanti che rendono più comprensivo il risultato ottenuto:

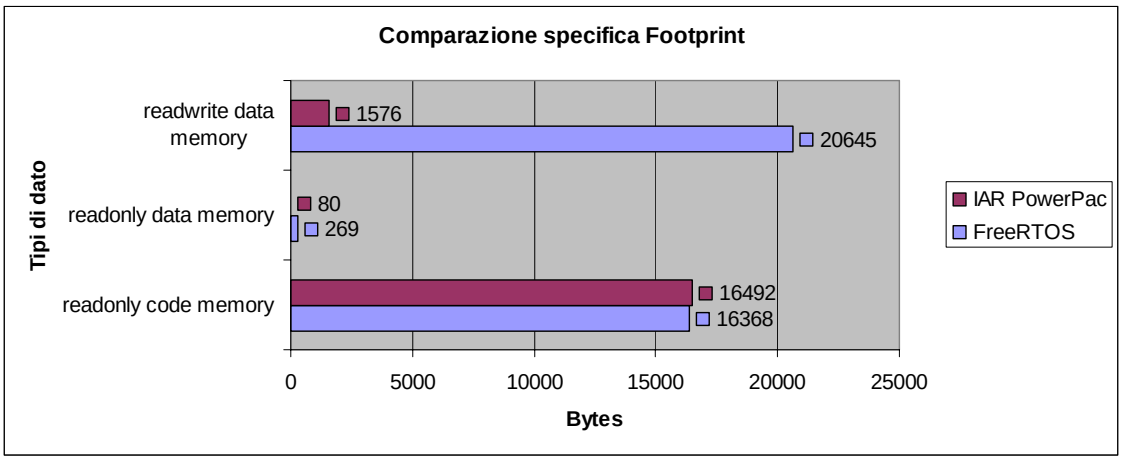


Fig. 6.4 Istogramma di comparazione tra i due RTOS relativamente al Footprint

SISTEMA OPERATIVO	TIPO DI DATO		
	readonly code memory	readonly data memory	readwrite data memory
FreeRTOS	16368 (bytes)	269 (bytes)	20645 (bytes)
IAR PowerPac	16492 (bytes)	80 (bytes)	1576 (bytes)

Tab. 6.2 Valori specifici relativi al Footprint dei due RTOS

Come si può vedere dal grafico e dalla tabella sovrastanti e come accennato al paragrafo 5.3.3, i risultati necessitano di alcuni commenti data la loro particolare varianza.
Innanzitutto è bene notare come l'occupazione di memoria totale sia stata suddivisa in tre parti potendo così effettuare un più accurato confronto tra i due RTOS; andiamo quindi ora per ogni tipo di dato presente sulle ordinate del grafico a vedere quale sia stata la sua effettiva occupazione in memoria.

Per quanto riguarda la memoria occupata dal codice (readonly code memory) vediamo come FreeRTOS abbia un leggero vantaggio; questo probabilmente è dovuto al fatto che IAR PowerPac implementi ulteriori funzionalità rispetto a FreeRTOS.
La memoria occupata dai dati in sola lettura (readonly data memory), ovvero le costanti, è molto ridotta per entrambi i sistemi; nello specifico, però, da notare come l'occupazione in memoria di IAR PowerPac sia inferiore di circa un fattore tre rispetto a quella di FreeRTOS.
Relativamente alla memoria occupata dalle variabili (readwrite data memory) il vantaggio è completamente ad appannaggio di IAR PowerPac in quanto la memoria dedicata ai dati in lettura/scrittura è inferiore di circa un fattore tredici rispetto al concorrente FreeRTOS.
Da qui è molto utile inserire un ulteriore istogramma individuante l'effettiva occupazione totale di memoria dei due RTOS ovvero il loro Footprint.

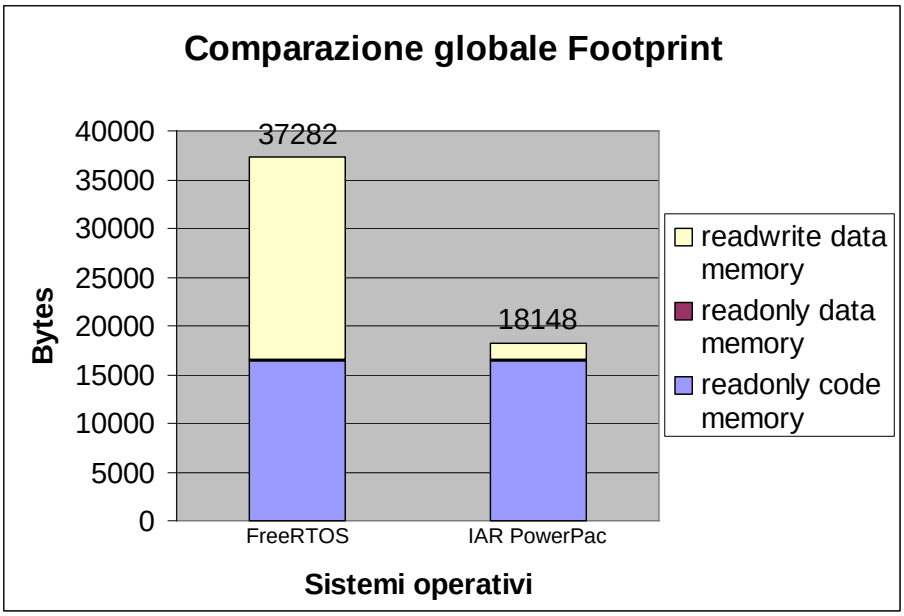


Fig. 6.5 Istogramma di comparazione globale relativamente al Footprint

Il grafico sopra rappresenta, per ciascuno dei due RTOS, la somma totale delle occupazioni in memoria relativamente ai vari tipi di dati sopraccitati e mostra, in maniera molto esplicita, come la memoria effettivamente occupata da IAR PowerPac sia circa la metà.
Si può ulteriormente evincere come, in condizioni di coerenza (stesso compilatore e stessa modalità di compilazione), IAR PowerPac abbia una performance in termini di Footprint di grande vantaggio rispetto a FreeRTOS.

6.2. Conclusioni e considerazioni

Tirando la rete in barca non ci resta che guardare quale sia il pesce migliore.

Entrambi i sistemi hanno caratteristiche di tutto rispetto ma ovviamente, come visto più volte in precedenza, ciascuno ha il proprio rovescio della medaglia; detto questo i risultati parlano chiaro poiché sia a livello qualitativo (Interrupt Latency) che a livello quantitativo (CST e Footprint), IAR PowerPac dimostra capacità prestazionali in termini di occupazione di risorse e di svolgimento dei compiti superiore a FreeRTOS.

Volendo effettuare una comparazione "forzata" ma allo stesso tempo assoluta relativamente all'assorbimento delle risorse da parte dei due sistemi si può giungere al grafico sottostante con la relativa tabella:

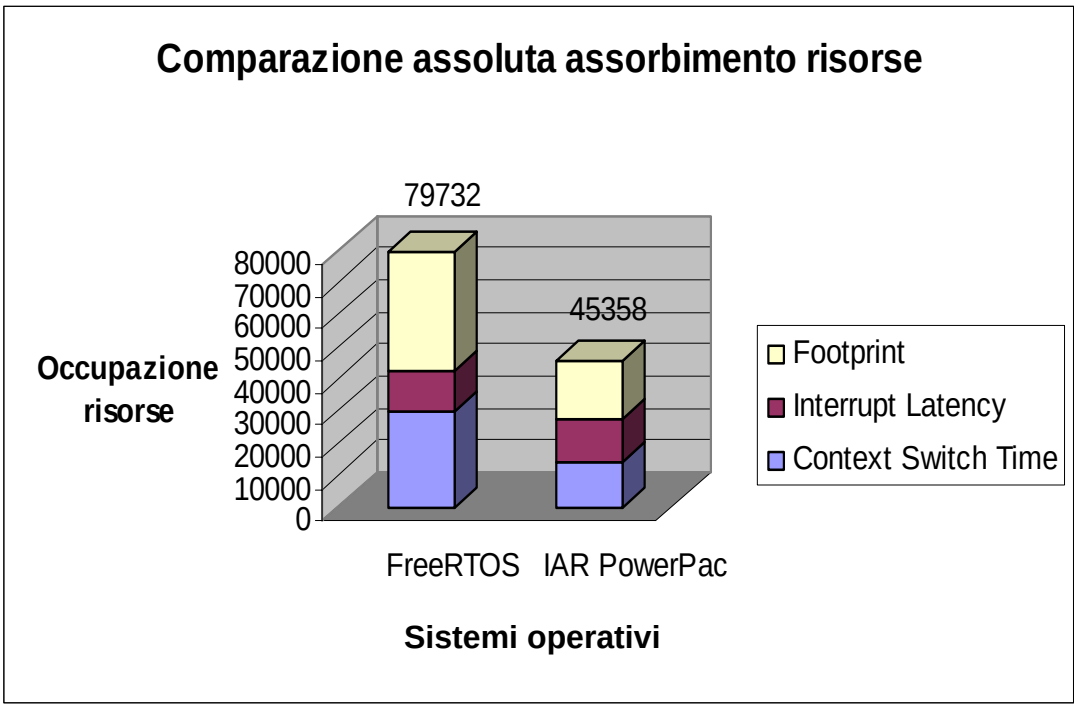


Fig. 6.6 Istogramma di comparazione assoluta tra i due RTOS

SISTEMA OPERATIVO	VALORI ASSOLUTAMENTE COMPARABILI			
	Context Switch Time	Interrupt Latency	Footprint	TOTALE
FreeRTOS	29450	13000	37282	79732
IAR PowerPac	14210	13000	18148	45358

Tab. 6.3 Valori specifici relativi alla comparazione assoluta tra i due RTOS

Per rendere in via grafica più facilmente visibile la conclusione che ne è sortita, è stata sviluppata una tabella utilizzando alcuni "artifizi" in maniera tale da costituire in corrispondenza ad essa un grafico ricco di significato.

Per far ciò innanzitutto abbiamo preso l'ordine di grandezza del Footprint come riferimento poiché numero intero ed abbiamo eliminato le unità di misura su tutti i parametri.

Dopodichè, per rendere graficamente comparabili e coerenti il CST e l'Interrupt Latency di entrambi i sistemi con l'ordine di grandezza del Footprint, abbiamo moltiplicato i valori del CST per un fattore 100 e quelli dell'Interrupt Latency per un fattore 1000.

Successivamente, per ogni sistema, abbiamo sommato i relativi valori diventati ormai numeri puri compatibili e coerenti ottenendo così un istogramma costituito da due barre comparabili rappresentanti l'assorbimento assoluto di risorse da parte dei due sistemi relativamente alla nostra architettura.

Per dare un senso grafico maggiormente intuitivo ai risultati provenienti dai vari tests svolti, inseriamo di seguito uno *spider plot* (detto anche grafico radar) nel quale vi sono rappresentati sui tre assi i parametri direttamente coinvolti nelle sessioni di test:

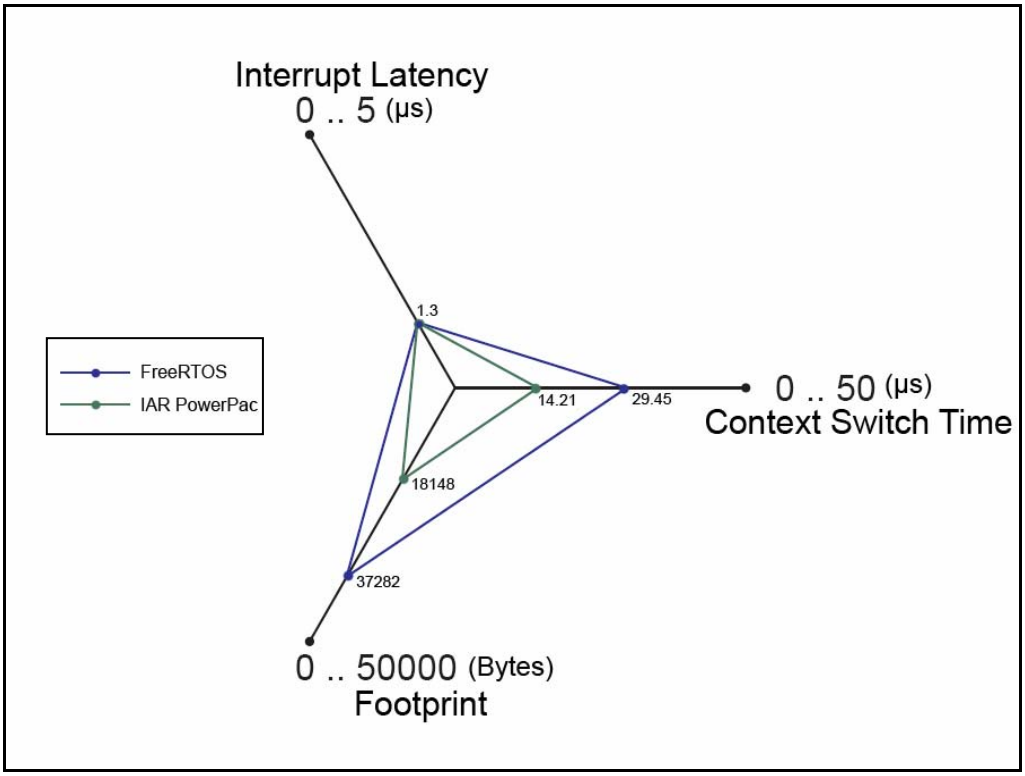


Fig. 6.7 Spider plot relativo all'assorbimento totale di risorse da parte dei due RTOS

Come si può evincere dalla rappresentazione grafica alla pagina precedente, le congiungenti tra le varie coppie di valori, disposte su assi adiacenti, generano forme triangolari (in questo caso due) abbinabili ciascuna di esse al relativo RTOS analizzato.
L'effettivo assorbimento assoluto di risorse da parte di ciascun RTOS è così rappresentato dall'area delimitata dal relativo triangolo prodotto nel grafico; in questo ambito è bene rammentare che il RTOS che ottiene il valore di area minore sia quello con prestazioni migliori rispetto all'altro.
Da qui possiamo affermare, dulcis in fundo, che IAR PowerPac sia "il sistema" più adatto ai nostri scopi operativi.

7. Eventuali sviluppi futuri

Eccoci dunque giunti alla fase conclusiva di questo elaborato.

Nel contesto tecnologico dei nostri tempi i dispositivi embedded hanno assunto un'importanza tale da essere contenuti in ormai qualunque dispositivo che necessiti di funzionare in maniera automatica; l'esplosione dell'utilizzo di questi dispositivi è dimostrata dalla notevole quantità di microchips embedded prodotti ma soprattutto dalla sconvolgente varietà di RTOS presenti sul mercato.

Avviamo quindi ora un breve percorso cronistorico che si snoderà dal ripercorrere i passaggi svolti da principio sino a quali migliorie e successivi sviluppi possano essere compiuti passando per lo stato attuale delle cose.

Abbiamo visto come siano state tanto numerose quanto differenti le fasi attraversate prima di poter decretare quale fosse il RTOS più adatto alle nostre esigenze; alla prima fase, che ci ha portati alla ricerca di informazioni in rete visitando i vari siti dedicati ai RTOS, è seguita un'analisi dei datasheet e delle tabelle riportanti i parametri prestazionali di tali sistemi. Dalle relative valutazioni sono stati individuati due sistemi operativi che, a giudicare dalle loro specifiche, sarebbero stati i reali candidati per un'ulteriore selezione finale che avrebbe individuato quale tra questi sarebbe diventato il RTOS favorito.

La nostra analisi, spinta dai nostri scopi specifici, si è svolta relativamente a tre parametri individuati assieme all'Ing. Matteo Cantarini; chiaramente, i parametri caratterizzanti un RTOS, sono molteplici e particolarmente poliedrici dando così al programmatore la sensazione di entrare in una fitta giungla dove ogni albero è un parametro.

Per quanto riguarda il nostro elaborato si limita all'analisi di un piccolo numero di parametri che però costituisce un ottima base per l'ampliamento dei test in maniera tale da poter così raggiungere una più completa valutazione dei RTOS in rapporto alla propria specifica applicazione e aver così una visione ottimale di tutte le possibili variabili in funzione della varianza dei parametri caratteristici che pilotano la scelta di tali sistemi.

Questo genere di paragoni, sempre giustificato dal fatto che i RTOS sono molteplici e con caratteristiche mutevoli, andrebbe compiuto ripetutamente a brevi intervalli di tempo mantenendo così in costante aggiornamento l'identità dei RTOS in base all'eventuale mutazione delle relative caratteristiche.

In questo contesto, come fase preliminare, abbiamo dovuto effettuare operazioni di porting; tali operazioni sono state effettuate con uno scopo prettamente d'importazione brutale dando così origine a pressoché impercettibili, ma inevitabili, rallentamenti nell'esecuzione del programma dovuti a porzioni di codice non particolarmente ottimizzate ma piuttosto atte a rendere possibile il corretto funzionamento dell'intero RTOS.
Da qui nasce naturale un ulteriore spunto che renderebbe interessante rianalizzare in seguito i valori sperimentali determinanti i RTOS dopo aver effettuato i dovuti "colpi di lima" al codice nativo.
Nei capitoli precedenti abbiamo ribadito più volte come sia di fondamentale importanza, durante lo svolgimento delle operazioni di testing, mantenere quanto più possibile la coerenza relativamente alla base ed alle ipotesi dei test stessi. Anche in questo caso uno spunto interessante nasce proprio da qui; sarebbe infatti molto utile ripetere le medesime procedure di testing su basi ed ipotesi differenti ma pur sempre coerenti così da poter valutare eventuali comportamenti funzionali non percettibili in una data sessione di test ma poter avere una visione di paragone tra i sistemi che possa offrire un quadro di confronto estremamente valido nella sua completezza.
In conclusione è bene comunque tener conto che al di là di tutti i test effettuabili su di un RTOS, resta come punto cardine fondamentale, il fatto che sia la scelta del RTOS che la scelta dei relativi parametri da testare varino in corrispondenza di come si abbia intenzione di utilizzare il RTOS stesso in quanto, come spesso recita il Prof. Ing. Rodolfo Zunino: "**chi comanda è sempre l'applicazione".**

Le applicazioni reali dell'architettura a cui si sta lavorando per questa piattaforma possono essere sintetizzate come sperimentazioni di applicazioni su scheda di valutazione di algoritmi per l'analisi di coordinate GPS.

8. Bibliografia

[1] Sistemi Elettronici Embedded
www.esng.dibe.unige.it/Staff/Personal-Pages/Rodolfo_Zunino/sitoDidattica/Download.htm
Rodolfo Zunino (DIBE UNIGE) - 2008

[2] Real-Time Operating Systems for DSP, part 6
www.dspdesignline.com/;jsessionid=VWXSRAAN5KQBGQSNDLRSKH0CJUNN2JVN
Robert Oshana – 2008

[3] Introduzione ai Microcontrollori
www.dei.unipd.it/~ieeesb/PIC/PresentazionePIC_01.pdf
IEEESB PADOVA – 2007

[4] RTOSBOOK
www.dti.unimi.it/~piuri/pages/didattica/SO/mat/rtoswebbook/
UNIMI – 2007

[5] Linux e sistemi embedded
www.ingpozzi.it/semlinux/presentazioni/slides_Ciminaghi.pdf
Davide Ciminaghi – 2004

[6] Scheduling della CPU
orfeo.unipv.it/cdol/luci_sisop/SO5.pdf
UNIPV – 2008

[7] Brevi Appunti sullo Scheduling
users.dimi.uniud.it/~alberto.casagrande/corsi/06-07/inf_gen/pdf/scheduling.pdf
UNIUD – 2007

[8] Solving starvation problems in the scheduler
lwn.net/Articles/176635/
Corbet – 2006

[9] The FreeRTOS.org Project
www.freertos.org/
FreeRTOS™

[10] RTXC Quadros™ Operating System
www.quadros.com/pdf/rtxc-quadros-datasheet.pdf
Quadros Systems, Inc. - 2008

[11] embOS: General Info
www.segger.com/embos_general.html
SEGGER – 2008

[12] ThreadX
www.rtos.com/page/product.php?id=2
Express Logic – 2008

[13] QNX Neutrino RTOS
www.qnx.com/products/neutrino_rtos/
QNX Software Systems – 2008

[14] LynxOS RTOS
www.lynuxworks.com/rtos/rtos.php
LYNUXWORKS – 2008

[15] IAR PowerPac™ for ARM
ftp.iar.se/WWWfiles/flyers/DS-PPARM-220.pdf
IAR – 2008

[16] A cosa serve il boundary-scan
www.strumentazioneelettronica.it/tecnologie/mr-volt/a-cosa-serve-il-boundary-scan-2008031935/
StrumentazioneElettornica – 2008

[17] LeCroy WaveRunner 6100A 2GHz 4ch Dual Oscilloscopi
www.computereq.com/detail.asp?ProdID=4124
COMPUTEREQ - 2008

[18] LPC2378 Preliminary Datasheet
www.kamami.pl/dl/lpc2378_datasheet_2006_10_11.pdf
NXP - 2006

[19] Il processore ARM
www.dei.unipd.it/corsi/ae1/store/x02aARMIntroduzione.pdf
DEI UNIPD – 2007

[20] Interfaccia Ethernet
www.dia.uniroma3.it/~necci/intro_rete.htm
Dipartimento di Informatica ed Automazione Universita' degli Studi "Roma Tre" – 2008

[21] Board per Centralina
www.visivagroup.it/showthread.php?t=8180&page=3
Visiva Group – 2006

[22] Introduzione alle UART
www.docmirror.net/it/linux/howto/networking/Modem-HOWTO/Modem-HOWTO-15.html
DOCMIRROR - 2008

[23] Section 15. Synchronous Serial Port (SSP)
ww1.microchip.com/downloads/en/DeviceDoc/31015a.pdf
Microchip – 1997

[24] ADC (Convertitori Analogico / Digitale)
www.galgani.it/sistemi_programmabili/analogico_digitale/adc/index_ita.asp
Pubblicazioni ed elaborati vari di Francesco Galgani – 2008

[25] Experiment 8 - Digital To Analog Conversion
www.learn-c.com/experiment8.htm
Controlling The Real World With Computers – 2008

[26] Introduction to Counter/Timer Hardware
www.netrino.com/Embedded-Systems/How-To/Timer-Counter-Unit
Netrino® and The Embedded Systems Experts℠ - 2008

[27] UM10211 LPC23xx User manual
www.nxp.com/acrobat_download/usermanuals/UM10211_1.pdf
NXP- 2008

[28] Calculate Interrupt Latency of FreeRTOS with LPC23xx
www.embeddedrelated.com/usenet/embedded/show/97784-1.php
Comp.Arch.Embedded - 2008